Java SE 7 Programmer I

O guia para sua certificação Oracle Certified Associate

GUIA DA CERTIFICAÇÃO

JAVA SE 7 • PROGRAMMER I • JAVA SE 7

GUILHERME SILVEIRA
MÁRIO AMARAL

Sumário

Capítulo 1

Agradecimentos

“Às três famílias que me acolhem no dia a dia, Azevedo Silveira, Bae Song e Caelum” - Guilherme Silveira

Escrever um livro é difícil, descrever pequenos detalhes de uma linguagem é um desafio maior do que poderíamos imaginar.

Fica um agradecimento ao Gabriel Ferreira, Márcio Marcelli, Leonardo Cordeiro e ao Alexandre Gamma pelas valiosas revisões dos textos e exercícios. Agradecimento especial ao Leonardo Wolter por sua revisão completa, além de diversas sugestões e melhorias.

Um abraço a todos da Caelum, do Alura e da Casa do Código, que nos incentivam na busca contínua de conhecimento com a finalidade de melhoria da qualidade de ensino e aprendizado de desenvolvimento de software no Brasil.

Capítulo 2

Certificação?

As certificações Java são, pelo bem ou pelo mal, muito reconhecidas no mercado. Em sua última versão, a principal certificação foi quebrada em duas provas. Este livro vai guiá-lo por questões e assuntos abordados para a primeira prova, a Java SE 7 Programmer I, de maneira profunda e desafiadora.

O livro vai percorrer cada tema, com detalhes e exercícios, para você chegar à prova confiante. Decorar regras seria uma maneira de estudar, mas não estimulante. Por que não compila? Por que não executa como esperado? Mais do que um guia para que você tenha sucesso na prova, nossa intenção é mostrar como a linguagem funciona por trás.

Ao terminar essa longa caminhada, você será capaz de entender melhor a linguagem, assim como poder dizer com exatidão os motivos de determinadas construções e idiomismos.

Como estudar

Lembre-se de usar a linha de comando do Java, não use o Eclipse ou qualquer outra IDE: os erros que o compilador da linha de comando mostra podem ser diferentes do da IDE, e você não quer que isso atrapalhe seu desempenho.

Lembre-se de ficar atento, na prova não ficará claro qual o assunto que está sendo testado e você deve se concentrar em todo o código, não só em um assunto ou outro.

Esse processo é longo e a recomendação é que agende a prova agora mesmo no site da Oracle, para que não haja pausa desde o primeiro dia de leitura, até o último dia de leitura, a execução de diversos simulados e a prova em si.

Não deixe de testar todo o código em que não sentir confiança. Os exercícios são gerados de propósito para causar insegurança no candidato, para levá-lo para um lado, sendo que o problema pode estar em outro. E faça muitos exercícios e simulados.

Não hesite, tire suas dúvidas no site do GUJ e nos avise de sua certificação via twitter ou facebook:

http://www.guj.com.br http://www.twitter.com/casadocodigo http://www.facebook.com/casadocodigo

Bom estudo, boa prova, boa sorte e, acima de tudo, bem-vindo ao grupo daqueles que não só usam uma linguagem, mas a dominam.

Seções da prova

Os assuntos cobrados e abordados aqui são:

1) Java Basics

- Define the scope of variables
- Define the structure of a Java class
- Create executable Java applications with a main method
- Import other Java packages to make them accessible in your code

2) Working With Java Data Types

- Declare and initialize variables
- Differentiate between object reference variables and primitive variables
- Read or write to object fields
- Explain an Object's Lifecycle (creation, "dereference" and garbage collection)
- Call methods on objects
- Manipulate data using the StringBuilder class and its methods
- Creating and manipulating Strings

3) Using Operators and Decision Constructs

- Use Java operators
- Use parenthesis to override operator precedence
- Test equality between Strings and other objects using == and equals ()
- Create if and if/else constructs
- Use a switch statement

4) Creating and Using Arrays

- Declare, instantiate, initialize and use a one-dimensional array
- Declare, instantiate, initialize and use multi-dimensional array
- Declare and use an ArrayList

5) Using Loop Constructs

- Create and use while loops
- Create and use for loops including the enhanced for loop
- Create and use do/while loops

- Compare loop constructs
- Use break and continue

6) Working with Methods and Encapsulation

    - Create methods with arguments and return values
    - Apply the static keyword to methods and fields
    - Create an overloaded method
    - Differentiate between default and user defined constructors
    - Create and overload constructors
    - Apply access modifiers
    - Apply encapsulation principles to a class
    - Determine the effect upon object references and primitive values when they are passed into methods that change the values

7) Working with Inheritance

    - Implement inheritance
    - Develop code that demonstrates the use of polymorphism
    - Differentiate between the type of a reference and the type of an object
    - Determine when casting is necessary
    - Use super and this to access objects and constructors
    - Use abstract classes and interfaces

8) Handling Exceptions

    - Differentiate among checked exceptions, RuntimeExceptions and Errors
    - Create a try-catch block and determine how exceptions alter normal program flow
    - Describe what Exceptions are used for in Java
    - Invoke a method that throws an exception
    - Recognize common exception classes and categories

Capítulo 3

O básico de Java

3.1 Defina o escopo de variáveis

O escopo é o que determina em que pontos do código uma variável pode ser usada.

Variáveis locais

Chamamos de locais as variáveis declaradas dentro de métodos ou construtores. Antes de continuar, vamos estabelecer uma regra básica: o ciclo de vida de uma variável local vai do ponto onde ela foi declarada até o fim do **bloco** onde ela foi declarada.

Mas o que é um bloco? Podemos entender como bloco um trecho de código entre chaves. Pode ser um método, um construtor, o corpo de um `if`, de um `for` etc.:

```
public void m1() { // início do bloco do método
    int x = 10; // variável local do método

    if (x >= 10) { // início do bloco do if
        int y = 50; // variável local do if
        System.out.print(y);

    } // fim do bloco do if

} // fim do bloco do método
```

Analisando esse código, temos uma variável `x`, que é declarada no começo do método. Ela pode ser utilizada durante todo o corpo do método. Dentro do `if`, declaramos a variável `y`. `y` só pode ser utilizada dentro do corpo do `if`, delimitado pelas chaves. Se tentarmos usar `y` fora do corpo do `if`, teremos um erro de compilação, pois a variável saiu do seu escopo.

Tome cuidado especial com loops `for`. As variáveis declaradas na área de inicialização do loop só podem ser usadas no corpo do loop:

```
for (int i = 0, j = 0; i < 10; i++)
    j++;

System.out.println(j); // erro, já não está mais no escopo
```

Parâmetros de métodos também podem ser considerados variáveis locais ao método, ou seja, só podem ser usados dentro do método onde foram declarados:

```
class Teste {

    public void m1(String bla) {
        System.out.print(bla);
    }

    public void m2() {
        // erro de compilação pois bla não existe neste
        // escopo
        System.out.println(bla);
```

```
    }
}
```

Variáveis de instância

Variáveis de instância ou variáveis de objeto são os atributos dos objetos. São declaradas dentro da classe, mas fora de qualquer método ou construtor. Podem ser acessadas por qualquer membro da classe e ficam em escopo enquanto o objeto existir:

```
class Pessoa {
    // variável de instância ou variável de objeto
    String nome;

    public void setNome(String n) {
        // acessando a variável de instância no método
        this.nome = n;
    }
}
```

Variáveis estáticas (class variables)

Podemos declarar variáveis que são compartilhadas por todas as instâncias de uma classe usando a palavra chave `static`. Essas variáveis estão no escopo da classe, e lá ficarão enquanto a classe estiver carregada na memória (enquanto o programa estiver rodando, na grande maioria dos casos).

```
class Pessoa {
    static int id = 1;
}

class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        System.out.println(p.id); // acessando pelo objeto
        System.out.println(Pessoa.id); // acessando direto pela
                                       // classe
    }
}
```

No caso de variáveis `static`, não precisamos ter uma referência para usá-las e podemos acessá-las diretamente a partir da classe, desde que respeitando as regras de visibilidade da variável.

Variáveis com o mesmo nome

Logicamente, não é possível declarar duas variáveis no mesmo escopo com o mesmo nome:

```
public void bla() {
    int a = 0;
    int a = 10;// erro
}
```

Mas, eventualmente, podemos ter variáveis em escopos diferentes que podem ser declaradas com o mesmo nome. Em casos em que possa haver ambiguidade na hora de declará-las, o próprio compilador irá emitir um erro evitando a confusão. Por exemplo, não podemos declarar variáveis de classe e de instância com o mesmo nome:

```
class Bla {
    static int a;
    int a; // erro de compilação,
}
...

System.out.println(new Bla().a); // qual variável estamos
                                 // acessando?
```

Também não podemos declarar variáveis locais com o mesmo nome de parâmetros:

```
public void metodo(String par) {
    int par = 0; // erro de compilação

    System.out.println(par); // qual?
}
```

Apesar de parecer estranho, é permitido declarar variáveis locais ou parâmetros com o mesmo nome de variáveis de instância ou de classe. Essa

técnica é chamada de *shadowing*. Nesses casos, é possível resolver a ambiguidade: para variáveis de classe, podemos referenciar pela própria classe; para variáveis de instância, usamos a palavra chave `this`:

```
class Pessoa {

    static int x = 0;
    int y = 0;

    public static void setX(int x) {
        // Usando a referência da classe
        Pessoa.x = x;
    }

    public void setY(int y) {
        // usando o this
        this.y = y;
    }
}
```

Quando não usamos o `this` ou o nome da classe para usar a variável, o compilador sempre utilizará a variável de menor escopo:

```
class X {
    int a = 10;

    public void metodo() {
        int a = 20; // shadowing
        System.out.println(a); // imprime 20
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
1 class Teste {
2     public static void main(String[] args) {
3         for (int i = 0; i < 20; i++) {
4             System.out.println(i);
5         }
```

```
6           int i = 15;
7           System.out.println(i);
8       }
9   }
```

a) Erro de compilação na linha 6. A variável `i` não pode ser redeclarada.

b) Erro de compilação na linha 7. A variável `i` é ambígua.

c) Compila e roda, imprimindo de 0 até 19 e depois 15.

d) Compila e roda, imprimindo de 0 até 19, depois ocorre um erro de execução na linha 6.

e) Compila e roda, imprimindo de 0 até 19 e depois 19 novamente.

2) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
1   class Teste {
2       static int x = 15;
3
4       public static void main(String[] x) {
5           x = 200;
6           System.out.println(x);
7       }
8   }
```

a) O código compila e roda, imprimindo 200.

b) O código compila e roda, imprimindo 15.

c) O código não compila.

d) O código compila mas dá erro em execução.

3) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
1   class Teste {
2       static int i = 3;
3
4       public static void main(String[] a) {
5           for (new Teste().i = 10; new Teste().i < 100;
```

```
6               new Teste().i++) {
7              System.out.println(i);
8          }
9      }
10 }
```

a) Não compila a linha 4.

b) Não compila a linha 5.

c) Compila e imprime 100 vezes o número 3.

d) Compila e imprime os números de 10 até 99.

3.2 Defina a estrutura de uma classe Java

Nesta seção, iremos entender a estrutura de um arquivo java, onde inserir as declarações de pacotes e imports e como declarar classes e interfaces.

Para entender a estrutura de uma classe, vamos ver o arquivo `Pessoa.java`:

```
1 // Declaração de pacote
2 package br.com.caelum.certificacao;
3
4 // imports
5 import java.util.Date;
6
7 // Declaração da classe
8 class Pessoa {
9     // conteúdo da classe
10 }
```

Pacotes

Pacotes servem para separar e organizar as diversas classes que temos em nossos sistemas. Todas as classes pertencem a um pacote, sendo que, caso o pacote não seja explicitamente declarado, a classe fará parte do que chamamos de **pacote padrão**, ou **default package**. Todas as classes no *default package* se enxergam e podem ser utilizadas entre si. Classes no pacote default não podem ser importadas para uso em outros pacotes:

```
// Uma classe no pacote padrão
classe Pessoa {
    //...
}
```

Para definir qual o pacote a que a classe pertence, usamos a palavra-chave `package`, seguida do nome do pacote. Só pode existir um único `package` definido por arquivo, e ele deve ser a primeira instrução do arquivo. Após a definição do `package`, devemos finalizar a instrução com um `;`. Podem existir comentários antes da definição de um pacote:

```
// declaração do pacote
package br.com.caelum.certificacao;

classe Pessoa {
    //...
}
```

Aproveitando que tocamos no assunto, o `package` deve ser a primeira instrução de código que temos declarada em nosso arquivo. Comentários não são considerados parte do código, portanto, podem existir em qualquer lugar do arquivo java sem restrições.

Para inserir comentário em nosso código, temos as seguintes formas:

```
// comentário de linha

/*
   comentário de
   multiplas linhas
 */
class /* comentário no meio da linha */ Pessoa {

    /**
     *  JavaDoc, repare que a primeira linha do comentário tem
     *  2 asteriscos
     */
    public void metodo() {
    }
}
```

Para saber mais: JavaDoc

Javadoc é um tipo especial de comentário que pode ser utilizado para gerar uma documentação HTML a partir de nosso código. Para saber mais, acesse http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html

Classe

Uma classe é a forma no Java onde definimos os atributos e comportamentos de um objeto. A declaração de uma classe pode ser bem simples, apenas a palavra `class` seguida do nome e de `{}`:

```
1 class Pessoa {}
```

Existem outros modificadores que podem ser usados na definição de uma classe, mas veremos essas outras opções mais à frente, onde discutiremos esses modificadores com mais detalhes.

Vale lembrar que java é *case sensitive* e `Class` é o nome de uma classe e não podemos usá-lo para definir uma nova classe.

Dentro de uma classe, podemos ter variáveis, métodos e construtores. Essas estruturas são chamadas de **membros** da classe.:

```
1 class Pessoa {
2
3     String nome;
4     String sobrenome;
5
6     Pessoa(String nome, String sobrenome) {
7         this.nome = nome;
8         this.sobrenome = sobrenome;
9     }
10
11    public String getNomeCompleto() {
12        return this.nome + this.sobrenome;
13    }
14 }
```

Nomes dos membros

Podemos ter membros de tipos diferentes com o mesmo nome. Fique atento, o código a seguir compila normalmente:

```
class B {
    String b;

    B() {
    }

    String b() {
        return null;
    }
}
```

Variaveis

Usando como exemplo a classe `Pessoa` definida anteriormente, `nome` e `sobrenome` são variáveis. A declaração de variáveis é bem simples, sempre o **tipo** seguido do **nome** da variável.

Dizemos que essas são variáveis de instância, pois existe uma cópia delas para cada objeto `Pessoa` criado em nosso programa. Cada cópia guarda o estado de uma certa instância desses objetos.

Existem ainda variáveis que não guardam valores ou referências para uma determinada instância, mas sim um valor compartilhado por todas as instâncias de objetos. Essas são variáveis **estáticas**, definidas com a palavra-chave `static`. Veremos mais sobre esse tipo de membro mais à frente.

Métodos

A declaração de métodos é um pouquinho diferente pois precisamos do **tipo do retorno**, seguido do **nome do método** e seguido de parênteses, sendo que pode ou não haver parâmetros de entrada desse método. Cada parâmetro é uma declaração de variável em si. Essa linha do método, onde está definido

o retorno, o nome e os parâmetros é onde temos a **assinatura do método**. Cuidado, pois a **assinatura de um método** inclui somente o nome do método e os tipos dos parâmetros.

Assim como variáveis, métodos também podem ser `static`, como veremos mais adiante.

Construtores

Uma classe pode possuir zero ou vários construtores. Nossa classe `Pessoa` possui um construtor que recebe como parâmetros o nome e o sobrenome da pessoa. A principal diferença entre a declaração de um método e um construtor é que um **construtor não tem retorno e possui o mesmo nome da classe**.

Métodos com o mesmo nome da classe

Cuidados com métodos que parecem construtores:

```
class Executa {

    // construtor
    Executa() {
    }

    // método
    void Executa() {
    }

}
```

Note que um construtor pode ter um `return` vazio:

```
class X {
    int j = -100;

    X(int i) {
```

```
5         if (i > 1)
6             return;
7         j = i;
8     }
9 }
```

Caso o valor seja menor ou igual a `1`, o valor de `j` será `-100`, caso contrário, será o mesmo valor de `i`.

Interfaces

Além de classes, também podemos declarar *interfaces* em nossos arquivos java. Para definir uma interface usamos a palavra reservada `interface`:

```
1 interface Autenticavel {
2 
3     final int TAMANHO_SENHA = 8;
4 
5     void autentica(String login, String senha);
6 }
```

Em uma interface, devemos apenas definir a assinatura do método, sem a sua implementação. Além da assinatura de métodos, também é possível declarar constantes em interfaces.

Multíplas estruturas em um arquivo

Em java, é possível definir mais de uma classe/interface em um mesmo arquivo java, embora devamos seguir algumas regras:

- Podem ser definidos em qualquer ordem;
- Se existir alguma classe/interface pública, o nome do arquivo deve ser o mesmo dessa classe/interface;
- Só pode existir uma classe/interface pública por arquivo;
- Se não houver nenhuma classe/interface pública, o arquivo pode ter qualquer nome.

Logo, são válidos:

```
// arquivo1.java
interface Bar {}

class Foo {}
```

```
// Foo.java
public class Foo {}

interface X {}
```

Pacotes e imports em arquivos com múltiplas estruturas

As regras de pacotes e imports valem também para arquivos com múltiplas estruturas definidas. Caso exista a definição de um pacote, ela vale para todas as classes/interfaces definidas nesse arquivo, e o mesmo vale para *imports*.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir sem nenhum parâmetro na linha de comando, como `java D`:

```
package a.b.c;

import java.util.*;

class D {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<String>();

        for (String arg : args) {
            if (new E().existe(arg))
                lista.add(arg);
        }
    }
```

```
14 }
15
16 import java.io.*;
17
18 class E {
19     public boolean existe(String nome) {
20         File f = new File(nome);
21         return f.exists();
22     }
23 }
```

a) O arquivo não compila.

b) O arquivo compila mas dá erro de execução pois o array é nulo.

c) O arquivo compila mas dá erro de execução pois o array tem tamanho zero.

d) Roda e imprime `false`.

e) Roda e imprime `true`.

f) Roda e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class Teste {
2      int Teste = 305;
3
4      void Teste() {
5          System.out.println(Teste);
6      }
7
8      public static void main(String[] args) {
9          new Teste();
10     }
11 }
```

a) O código não compila: erros nas linhas 24, 25 e 26.

b) O código não compila: erro na linha 25.

c) O código não compila: erros nas linhas 24 e 26.

d) O código compila e, ao rodar, imprime `305`.

e) O código compila e não imprime nada.

f) O código compila e, ao rodar, imprime uma linha em branco.

3) Escolha a opção adequada ao tentar compilar o arquivo a seguir:

```
1 package br.com.teste;
2
3 import java.util.ArrayList;
```

a) Erro na linha 1: definimos o pacote mas nenhum tipo.

b) Erro na linha 3: importamos algo desnecessário ao arquivo.

c) Compila sem erros.

4) Escolha a opção adequada ao tentar compilar o arquivo `A.java`:

```
1 class A implements B {
2 }
3 public interface B {
4 }
5 class C extends A {
6 }
7 class D extends A, implements B {
8 }
```

a) Não compila: erro na linha 7.

b) Não compila: erro na linha 1.

c) Não compila: erro na linha 1, 5 e 7.

d) Não compila: erro na linha 3.

e) Compila.

3.3 Crie aplicações Java executáveis com um método main

Nesta seção, entenderemos as diferenças entre classes normais e classes que podem ser executadas pela linha de comando.

Uma classe executável é uma classe que possui um método inicial para a execução do programa o método `main`, que será chamado pela JVM. Classes sem o método `main` não são classes executáveis e não podem ser usadas como ponto inicial da aplicação.

Método main

O tal método de entrada deve seguir algumas regras para ser executado pela JVM:

- Ser público (`public`);
- Ser estático (`static`);
- Não ter retorno (`void`);
- Ter o nome *main*;
- Receber como parâmetro um `array` ou `varargs` de String (`String[]` ou `String...`).

São então métodos `main` válidos os seguintes exemplos:

```
//Parâmetro como array
public static void main (String[] args) {}

//Parâmetro como varargs
public static void main (String... args) {}

//A ordem dos modificadores não importa
static public void main(String[] args) {}

//O nome do parâmetro não importa
public static void main (String... argumentos){}
```

```
12
13 //Também é uma definição válida de array
14 public static void main (String args[]) {}
```

Executando uma classe pela linha de comando

Para executar uma classe com `main` pela linha de comando, devemos compilar o arquivo com o comando `javac` e executar a classe com o comando `java`:

Usando o arquivo `HelloWorld.java` a seguir:

```
1 public class HelloWorld {
2
3     public static void main(String[] args) {
4         System.out.println("Hello World! ");
5     }
6 }
```

Compilamos e executamos no terminal com os seguintes comandos:

```
$ javac HelloWorld.java
$
$ java HelloWorld
Hello World!
```

Repare que, para compilar a classe, passamos como parâmetro para o comando `javac` o nome do arquivo, enquanto para executar, passamos apenas o nome da classe (`HelloWorld`) para o comando `java`.

Passando parâmetros pelo linha de comando

Ao executarmos uma classe pela linha de comando, podemos passar parâmetros para o método `main`. Esses valores serão recebidos no array do método `main`. Por exemplo, vamos passar um nome para a classe `HelloWorld`:

```
1 public class HelloWorld{
2
3     public static void main(String[] args) {
```

```
4        //Lendo o valor da primeira posição do array args
5        System.out.println("Hello " + args[0] + "!");
6    }
7 }
```

Para informar o valor do parâmetro, é só informá-lo **APÓS** o nome da classe que está sendo executada:

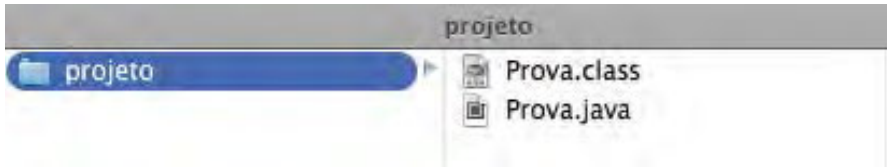
```
java HelloWorld Mario
Hello Mario!
```

Você pode passar quantos parâmetros quiser, basta separá-los por espaço. Cada parâmetro informado será armazenado em uma posição do array, na mesma ordem em que foi informado.

Compilação e execução

Para criar um programa java, é preciso escrever um código-fonte e, através de um compilador, gerar o executável (bytecode). O compilador do JDK (*Java Development Kit*) é o *javac*. Para a prova de certificação, devemos conhecer o comportamento desse compilador.

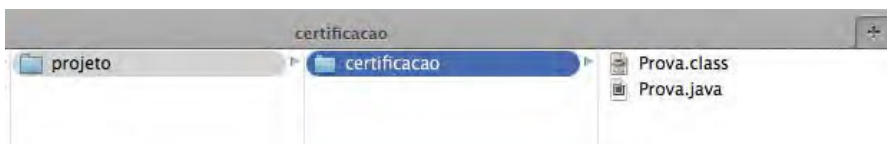
A execução do bytecode é feita pela **JVM** (*Java Virtual Machine*). O comando `java` invoca a máquina virtual para executar um programa java. Ao baixarmos o Java, podemos escolher baixar o JDK, que já vem com o JRE, ou somente o JRE (*Java Runtime Environment*), que inclui a *Virtual Machine*.

Algumas questões da prova abordam aspectos fundamentais do processo de compilação e de execução. É necessário saber como os comandos `javac` e o `java` procuram os arquivos.

javac

Imagine o arquivo `Prova.java` dentro do diretório de meu projeto:

```
class Prova {
    double tempo;
}
```

```
$ javac Prova.java
```

O bytecode da classe `Prova` gerado na compilação é colocado no arquivo `Prova.class` dentro do nosso diretório de trabalho, no meu caso, `projeto`. O resultado é:

Os projetos profissionais utilizam o recurso de pacotes para melhor organizar os fontes e os bytecodes. Vejamos qual é o comportamento do `javac` com a utilização de pacotes. Colocamos o arquivo `Prova.java` no diretório `certificacao`:

```
package certificacao;
class Prova {
    double tempo;
}
```

```
[certificacao]$ javac certificacao/Prova.java
```

Nesse exemplo, o arquivo `Prova.class` é colocado no diretório `certificacao`.

> **Escolhendo a versão do Java na hora de compilar**
>
> Na hora da compilação, é possível definir em que versão do Java o código-fonte foi escrito. Isso é feito com a opção `-source` do comando `javac`. (`javac MinhaClasse.java -source 1.3`).

java

Vamos utilizar um exemplo para mostrar o funcionamento do comando `java`, criando o arquivo `Teste.java` no mesmo diretório, no mesmo pacote:

```
package certificacao;
class Teste {
    public static void main(String[] args) {
        Prova p = new Prova();
        p.tempo = 210;
        System.out.println(p.tempo);
    }
}
```

```
$ javac certificacao/Teste.java
$ java certificacao.Teste
```

Saída:

```
210.0
```

E o resultado são os arquivos:

Somente o arquivo `Teste.java` foi passado para o compilador. Nesse arquivo, a classe `Teste` utiliza a classe `Prova` que se encontra em outro arquivo, `Prova.java`. Dessa forma, o compilador vai compilar automaticamente o arquivo `Prova.java` se necessário.

Para executar, é preciso passar o nome completo da classe desejada para a máquina virtual. O sufixo `.class` não faz parte do nome da classe, então ele não aparece na invocação da máquina virtual pelo comando `java`.

Propriedades na linha de comando

A prova ainda cobra conhecimentos sobre como executar um programa java passando **parâmetros** ou **propriedades** para a JVM e essas propriedades são identificadas pelo *-D* antes delas. **Este *-D* não faz parte da chave**.

```
java -Dchave1=abc -Dchave2=def Foo xpto bar
```

`chave1=abc` e `chave2=def` são parâmetros/propriedades e `xpto` e `bar` são argumentos recebidos pelo método `main`.

Classpath

Para compilar ou para executar, é necessário que os comandos `javac` e `java` possam encontrar as classes referenciadas pela aplicação java.

A prova de certificação exige o conhecimento do algoritmo de busca das classes. As classes feitas pelo programador são encontradas através do **classpath** (caminho das classes).

O classpath é formado por **diretórios**, **jars** e **zips** que contenham as classes e pacotes da nossa aplicação. Por padrão, o classpath está configurado para o diretório corrente (`.`).

Configurando o classpath

Há duas maneiras de configurar o classpath:

1) Configurando a variável de ambiente `CLASSPATH` no sistema operacional.

Basta seguir as opções do SO em questão e definir a variável. Isso é considerado uma má prática no dia a dia porque é um classpath global, que vai valer para qualquer programa java executado na máquina.

2) Com as opções `-cp` ou `-classpath` dos comandos `javac` ou `java`.

É a forma mais usada. Imagine que queremos usar alguma biblioteca junto com nosso programa:

```
$ javac -cp /diretorio/biblioteca.jar Prova.java
$ java -cp /diretorio/biblioteca.jar Prova
```

E podemos passar tanto caminhos de outras pastas como de `JARs` ou `zips`. Para passar mais de uma coisa no classpath, usamos o separador de parâmetros no SO (no Windows é ponto e vírgula, no Linux/Mac/Solaris/Unix são dois pontos):

```
$ javac -cp /diretorio/biblioteca.jar;/outrodir/ scjp/Prova.java
$ java -cp /diretorio/biblioteca.jar;/outrodir/ scjp.Prova
```

Para saber mais: arquivos JAR

Para facilitar a distribuição de bibliotecas de classes ou de aplicativos, o JDK disponibiliza uma ferramenta para a compactação das classes java.

Um arquivo `JAR` nada mais é que a pasta de nossas classes no formato `ZIP` mas com extensão `.jar`.

Para criar um `jar` incluindo a pasta `scjp` que fizemos antes:

```
jar -cf bib.jar scjp
```

Agora podemos executar nossa classe usando esse `jar`:

```
java -cp bib.jar scjp.Prova
```

Para saber mais: META-INF/Manifest.mf

Ao criar o jar usando o comando `jar` do JDK, ele cria automaticamente a pasta `META-INF`, que é usada para configurações relativas ao nosso `jar`. E dentro dela, cria o arquivo `Manifest.mf`.

Esse arquivo pode ser usado para algumas configurações. Por exemplo, é possível dizer qual classe do nosso `jar` é a classe principal (**Main-Class**) e que deve ser executada.

Basta criar um arquivo chamado `Manifest.mf` com a seguinte instrução indicando a classe com o método `main`:

```
Main-Class: scjp.Teste
```

E depois gerar o jar passando esse arquivo:

```
jar -cfm bib.jar meumanifest scjp
```

Na hora de rodar um jar com `Main-Class`, basta usar:

```
java -jar bib.jar
```

1) Qual é uma assinatura válida do método `main` para executar um programa java?

    a) `public static void main(String... args)`
    b) `public static int main(String[] args)`
    c) `public static Void main(String []args)`
    d) `protected static void main(String[] args)`
    e) `public static void main(int argc, String[] args)`

2) Escolha a opção adequada para compilar e rodar o arquivo `A.java`, existente no diretório `b`:

```
1 package b;
2 class A {
3     public static void main(String[] args) {
4         System.out.println("rodando");
5     }
6 }
```

a) `javac A` e `java A`

b) `javac A.java` e `java A`

c) `javac b/A.java` e `java A`

d) `javac b/A.java` e `java b.A`

e) `javac b.A.java` e `java b.A`

f) `javac b.A` e `java b.A`

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         System.out.println(args);
4         System.out.println(args.length);
5         System.out.println(args[0]);
6     }
7 }
```

a) Não compila: array não possui membro `length`.

b) Não compila: o método `println` não consegue imprimir um array.

c) Ao rodar sem argumentos, ocorre uma `NullPointerException` na linha 5.

d) Ao rodar sem argumentos, ocorre uma `NullPointerException` na linha 4.

e) Ao rodar sem argumentos, são impressos os valores "1" e "A".

f) Ao rodar com o argumento "certificacao", são impressos os valores "2" e "A".

4) Escolha a opção adequada para rodar a classe `A.java` presente no diretório `b`, que foi compactado em um arquivo chamado `programa.jar`, sendo que não existe nenhum arquivo de manifesto:

```
1 package b;
2 class A {
3     public static void main(String[] args) {
4         System.out.println(args[0]);
5     }
6 }
```

a) `java jar programa.jar`

b) `java jar programa.jar b.A`

c) `java -jar programa.jar`

d) `java -jar programa.jar b.A`

e) `java -cp programa.jar`

f) `java -cp programa.jar b.A`

5) Escolha a opção adequada para compilar a classe `A.java`, definida como no pacote `b` presente no diretório `b`, e adicionar também o arquivo `programa.jar` na busca de classes durante a compilação. Lembre-se que `.` significa o diretório atual.

a) `javac -cp b.A.java -cp programa.jar`

b) `javac -jar programa.jar b.A.java`

c) `javac -cp programa.jar:b A.java`

d) `javac -cp programa.jar:. b.A.java`

e) `javac -cp . -cp programa.jar`

f) `javac -jar programa.jar:. b/A.java`

g) `javac -cp programa.jar:b b/A.java`

h) `javac -cp programa.jar:. b/A.java`

3.4 Importe outros pacotes Java e deixe-os acessíveis ao seu código

Se duas classes estão no mesmo pacote, elas se "enxergam" entre si, sem a necessidade de colocar o nome do pacote. Por exemplo, imagine que as classes `Pessoa` e `Endereco` estejam no mesmo pacote:

```
1 package modelo;
2 
3 class Endereco {
4     String rua;
5     String numero;
6     String bairro;
7     //...
8 }
```

E o outro arquivo:

```
1 package modelo;
2 
3 class Pessoa {
4     Endereco endereco; // Pessoa usando o endereço
5 }
```

Para usar uma classe que está em outro pacote, temos duas opções: podemos referenciá-la usando o que chamamos de `Full Qualified Name`, ou seja, o nome do pacote seguido do nome da classe. O código ficaria assim:

```
1 package financeiro;
2 
3 class Pedido {
4     modelo.Pessoa cliente; // Usando a classe Pessoa de outro
5                            // pacote
6 }
```

Tentamos compilar mas ele não deixa, porque uma classe, por padrão, só pode ser acessada dentro do próprio pacote, e a nossa classe `Pessoa` está no pacote `modelo`. Portanto, definiremos nossa classe `Pessoa` como pública. Veremos com mais calma os modificadores de acesso na seção que cobra isso

na prova. Por enquanto, basta lembrar que classes públicas podem ser acessadas por outros pacotes, já classes padrão não podem.

```
package modelo;

public class Pessoa {
    Endereco endereco // Pessoa usando o endereço
}
```

Outra opção é importar a classe `Produto` e referenciá-la apenas pelo nome simples dentro de nosso código. Para fazer o *import* usamos a palavra `import`, seguida do `Full Qualified Name` da classe. A instrução de `import` deve aparecer na classe logo após o `package` (se este existir), e antes da definição da classe. É possível importar mais de uma classe por vez:

```
package modelo;

// Importando a classe Produto do pacote estoque
import estoque.Produto;
// Outro import qualquer
import java.util.Date;

class Pedido {
    Pessoa cliente; // mesmo pacote
    Produto item; // importado
    Date dataEmissao; //importado
}
```

Também é possível importar todas as classes de um determinado pacote, basta usar um `*` após o nome do pacote:

```
// Importando todas as classes do pacote estoque
import estoque.*;
```

Importando classes com mesmo nome

Quando precisamos usar duas classes com o mesmo nome mas de pacotes diferentes, só podemos importar uma delas. A outra deve ser referenciada pelo `Full Qualified Name`. Tentativas de importar as duas classes irão resultar em erros de compilação:

```
1 import java.util.Date;
2 import java.sql.Date; // Erro de compilação pois temos duas
3                       // classes Date
4
5 class Teste {
6     Date d1;
7     Date d2;
8 }
```

O correto seria:

```
1 import java.util.Date;
2
3 class Teste {
4     Date d1;             // java.util
5     java.sql.Date d2; // java.sql
6 }
```

Caso tenhamos um import específico e um import genérico, o Java usa o específico:

```
import java.util.*;
import java.sql.Date;

class Teste{
    Date d1; // java.sql
    Date d2; // java.sql
}
```

Por padrão, todas as classes do pacote `java.lang` são importadas. Um ponto importante é que nenhuma classe de pacote que não seja o padrão pode importar uma classe do pacote padrão:

```
class Gerente {
}
```

```
package modelo;
classe Banco {
    Gerente gerente; // não compila pois não é possível importar
                     // tipos do pacote padrão de jeito *nenhum*
}
```

Pacotes

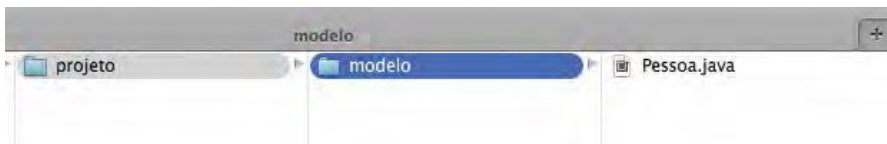
Nesta seção, entenderemos mais a fundo como funciona a declaração de pacotes, e como isso influencia nos imports das classes.

Como já discutimos anteriormente, pacotes servem para organizar suas classes e interfaces. Eles permitem agrupar componentes que tenham alguma relação entre si, além de garantir algum nível de controle de acesso a membros. Além de serem uma divisão lógica para as suas classes, os pacotes também definem uma separação física entre os arquivos de seu projeto, já que espelham a estrutura de diretórios dos arquivos do projeto.

Subpacotes e estrutura de diretórios

Pacotes são usados pela JVM como uma maneira de encontrar as classes no sistema de arquivos, logo a estrutura de diretórios do projeto deve ser a mesma da estrutura de pacotes. Vamos usar como exemplo a classe Pessoa:

```
1 package projeto.modelo;
2
3 public class Pessoa {}
```

O arquivo `Pessoa.java` deve estar localizado dentro do diretório `modelo`, que deve estar dentro do diretorio `projeto`, conforme a figura a seguir:

Dizemos que `modelo` é um subpacote de `projeto`, já que está dentro dele. Podemos ter vários subpacotes, como `projeto.utils` e `projeto.conversores`, por exemplo. Usamos o caractere `.` como separador de pacotes e subpacotes.

Convenções de nomes para pacotes

Existem algumas convenções para nomes de pacotes. Elas não são obrigatórias, mas geralmente são seguidas para facilitar o entendimento e organização do código:

- O nome do pacote deve ser todo em letras minúsculas;
- Um pacote deve começar com o site da empresa, ao contrário;
- Após o site, deve vir o projeto;
- Após o projeto, a estrutura é livre.

Import usando classes de outros pacotes

Existem diversas maneiras de referenciar uma classe de pacote diferente em nosso código. Vamos analisar essas opções:

Full Qualified Name

Podemos referenciar uma classe em nosso código usando o que chamamos de **Full Qualified Name**, ou **FQN**. Ele é composto pelo **pacote completo** mais o **nome da classe**, por exemplo:

```
1 class Pessoa {
2     // FQN da classe Calendar
3     java.util.Calendar dataDeNascimento;
4 }
```

import

Usar o `FQN` nem sempre deixa o código legível, portanto, em vez de usar o nome completo da classe, podemos importá-la e usar apenas o nome simples da classe:

```
1 import java.util.Calendar;
2
3 class Pessoa {
4     Calendar dataDeNascimento;
5 }
```

É permitido também importar todas as classes de um pacote de uma vez, usando o `*` no lugar do nome da classe:

```
import java.util.*;

class Pessoa {
    // Calendar e List são do pacote java.util
    Calendar dataDeNascimento;
    List<String> apelidos;
}
```

Caso existam duas classes com o mesmo nome, mas de pacotes diferentes, só podemos importar uma delas. A outra deve ser referenciada pelo `FQN`:

```
import java.util.Date;

class Foo {
    //do java.util
    Date some;
    java.sql.Date other;
}
```

]

Multiplos imports com *

Caso importemos dois ou mais pacotes que contenham classes com o mesmo nome, será obrigatório especificar, usando o `FQN`, qual das classes queremos utilizar. Ao tentar usar apenas o nome simples da classe, teremos um erro de compilação:

```
import java.util.*;
import java.sql.*;

public class Testes {
    private Date d; // Erro de compilação, de qual pacote é
                    // para usar?
}
```

Import de subpacotes

Em Java, não podemos importar todas as classes de subpacotes usando *. Veja a seguinte situação, considerando que cada classe foi definida em seu próprio arquivo:

```
1 package sistema.prova;
2
3 public class Pergunta {}
```

```
1 package sistema.banco;
2
3 public class PerguntaDao {}
```

```
1 package sistema;
2
3 public class Exame {}
```

```
1 package sistema.teste;
2
3 import sistema.*; //só importou a classe Exame
4
5 public class Teste {}
```

O único modo de importar todas as classes é explicitamente importando cada subpacote:

```
1 package sistema.teste;
2
3 import sistema.*;
4 import sistema.prova.*;
5 import sistema.banco.*;
6 public class Teste {}
```

import static

Desde o Java 5, é possível importar apenas métodos e atributos estáticos de uma classe, usando a palavra-chave `static` juntamente com o `import`. Podemos importar um a um ou simplesmente importar todos usando `*` :

```
package model;

public class Utils {

    // Atributo estático público
    public static int VALOR = 0;
    // Métodos estáticos públicos
    public static void metodo1() {}
    public static void metodo1(int a) {}

}
```

```
// Importando todos os membros public static de Utils
import static model.Utils.*;

public class Testes {

    public static void main(String[] args) {
        int x = VALOR;
        metodo1();
        metodo1(x);
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o `Teste`. Arquivo no diretório atual:

```
import modelo.Cliente;
class Teste {
    public static void main(String[] args) {
        new Cliente("guilherme").imprime();
    }
}
```

Arquivo no diretório `modelo`:

```
package modelo;

class Cliente {
```

```
4       private String nome;
5       Cliente(String nome) {
6           this.nome = nome;
7       }
8       public void imprime() {
9           System.out.println(nome);
10      }
11  }
```

a) Não compila: erro na classe `Teste`.

b) Não compila: erro na classe `Cliente`.

c) Erro de execução: método `main`.

d) Roda e imprime "Guilherme".

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  import modelo.basico.Cliente;
2  import modelo.avancado.Cliente;
3
4  class Teste {
5      public static void main(String[] args) {
6          System.out.println("Bem vindo!");
7      }
8  }
```

a) O código não compila, erro ao tentar importar duas classes com o mesmo nome.

b) O código compila, mas ao rodar dá erro por ter importado duas classes com o mesmo nome.

c) O código compila e roda imprimindo `Bem vindo!`, uma vez que nenhuma das classes importadas é usada no código, não existe ambiguidade.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sabendo que existem duas classes `Cliente`, uma no pacote `basico` e outra no pacote `avancado`:

```
1 import modelo.basico.Cliente;
2 import modelo.avancado.*;
3
4 class Teste {
5     public static void main(String[] args) {
6         System.out.println("Bem vindo!");
7     }
8 }
```

a) O código não compila, erro ao tentar importar duas classes com o mesmo nome.

b) O código compila mas ao rodar dá erro por ter importado duas classes com o mesmo nome.

c) O código compila e roda imprimindo `Bem vindo!`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import modelo.basico.Cliente;
2 import modelo.basico.Cliente;
3
4 class Teste {
5     public static void main(String[] args) {
6         System.out.println("Bem vindo!");
7     }
8 }
```

a) O código não compila, erro ao tentar importar duas classes com o mesmo nome.

b) O código compila, mas ao rodar dá erro por ter importado duas classes com o mesmo nome.

c) O código compila e roda imprimindo `Bem vindo!`, uma vez que não há ambiguidade.

5) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

`a/A.java`:

```
package a;
class A {
    b.B variavel;
}
```

a/C.java:

```
package a;
class C {
    b.B variavel;
}
```

a/b/B.java:

```
package a.b;
class B {
}
```

a) Erro de compilação somente no arquivo A.

b) Erro de compilação somente no arquivo B.

c) Erro de compilação somente no arquivo C.

d) Erro de compilação nos arquivos A e B.

e) Erro de compilação nos arquivos A e C.

f) Erro de compilação nos arquivos B e C.

g) Compila com sucesso.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package A;
class B{
    public static void main(String[] a) {
        System.out.println("rodei");
    }
}
```

a) Não compila: a variável do método `main` deve se chamar `args`.

b) Não compila: pacote com letra maiúscula.

c) Compila mas não roda: a classe B não é pública.

d) Compila e roda.

7) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

a/A.java:

```
package a;
public class A {
    public static final int VALOR = 15;
    public void executa(int x) {
        System.out.println(x);
    }
}
```

b/B.java:

```
package b;
import static a.A.*;
class B{
    void m() {
        A a = new A();
        a.executa(VALOR);
    }
}
```

a) B não compila: erro na linha 2.

b) B não compila: erro na linha 5.

c) B não compila: erro na linha 6.

d) Tudo compila.

8) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

a/A.java:

```
1 package a;
2 public class A {
3     public static final int VALOR = 15;
4     public void executa(int x) {
5         System.out.println(x);
6     }
7 }
```

`b/B.java`:

```
1 package b;
2 import a.A;
3 static import a.A.*;
4 class B{
5     void m() {
6         A a = new A();
7         a.executa(VALOR);
8     }
9 }
```

a) B não compila: erro na linha 3.

b) B não compila: erro na linha 5.

c) B não compila: erro na linha 6.

d) Tudo compila.

9) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

`A.java`:

```
1 public class A {
2     public static final int VALOR = 15;
3     public void executa(int x) {
4         System.out.println(x);
5     }
6 }
```

`b/B.java`:

```
1 package b;
2 import static A.*;
3 class B{
4     void m() {
5         A a = new A();
6         a.executa(VALOR);
7     }
8 }
```

a) Não compila

b) Tudo compila.

Capítulo 4

Trablhando com tipos de dados em Java

4.1 Declarar e inicializar variáveis

Qualquer programa de computador precisa manter informações de alguma forma. As linguagens de programação permitem a criação de variáveis para que possamos armazenar informações. Por exemplo, se precisarmos guardar a idade de uma pessoa, podemos utilizar uma variável que seja capaz de manter números inteiros.

Quando precisamos de uma nova variável, devemos declarar que queremos criá-la. A declaração de variável no Java, obrigatoriamente, deve informar o **tipo** e o **nome** que desejamos para ela. Por isso, essa linguagem é dita *explicitamente tipada* (todas as variáveis precisam ter o seu tipo definido).

```
// Declaração de uma variável chamada idade do tipo primitivo int
int idade;
```

Nem toda linguagem exige que as variáveis sejam iniciadas antes de serem utilizadas. Mas, no Java, a **inicialização é obrigatória** e pode ser implícita ou explícita. É de fundamental importância saber que, para usar uma variável, é necessário que ela tenha sido iniciada explicitamente ou implicitamente em algum momento antes da sua utilização.

Variáveis locais (declaradas dentro de métodos/construtores) devem ser iniciadas antes de serem utilizadas, ou teremos um erro de compilação:

```
// Declaração
int idade;

System.out.println(idade); // erro de compilação
```

```
// Declaração
int idade;

// Inicialização explícita de uma variável
idade = 10;

// Utilização da variável
System.out.println(idade); // ok
```

Podemos declarar e iniciar a variável na mesma instrução:

```
// Declaração e inicialização explícita na mesma linha
double pi = 3.14;
```

Se eu tenho um `if`, a inicialização deve ser feita em todos os caminhos possíveis:

```
void metodo(int a) {
    double x;
    if(a > 1) {
        x = 6;
    }
    System.out.println(x); // talvez x não tenha sido
                           // inicializado, portanto não compila
}
```

Quando a variável é membro de uma classe, ela é iniciada implicitamente junto com o objeto com um valor *default*:

```
class Prova {
    double tempo;
}

// Implicitamente, na criação de um objeto Prova,
// o atributo tempo é iniciado com 0
Prova prova = new Prova();

// Utilização do atributo tempo
System.out.println(prova.tempo);
```

Outro momento em que ocorre a inicialização implícita é na criação de arrays:

```
int[] numeros = new int[10];
System.out.println(numeros[0]); // imprime 0
```

Quando iniciadas implicitamente, os valores default para as variáveis são:

- primitivos numéricos inteiros **0**
- primitivos numéricos com ponto flutuante **0.0**
- `boolean` **false**
- `char` **vazio**, equivalente a 0
- referências **null**

Os tipos das variáveis do Java podem ser classificados em duas categorias: primitivos e não primitivos (referências).

Tipos primitivos

Todos os tipos primitivos do Java já estão definidos e não é possível criar novos tipos primitivos. São oito os tipos primitivos do Java: `byte`, `short`, `char`, `int`, `long`, `float`, `double` e `boolean`.

O `boolean` é o único primitivo não numérico. Todos os demais armazenam números: `double` e `float` são ponto flutuante, e os demais, todos inteiros (incluindo `char`). Apesar de representar um caractere, o tipo `char` armazena seu valor como um número positivo. Em Java, não é possível declarar variáveis com ou sem sinal (*unsigned*), todos os números (exceto `char`) podem ser positivos e negativos.

Cada tipo primitivo abrange um conjunto de valores. Por exemplo, o tipo `byte` abrange os números inteiros de -128 até 127. Isso depende do tamanho em bytes do tipo sendo usado.

Os tipos inteiros têm os seguintes tamanhos:

- `byte` 1 byte (8 bits, de -128 a 127);
- `short` 2 bytes (16 bits, de –32.768 a 32.767);
- `char` 2 bytes (só positivo), (16 bits, de 0 a 65.535);
- `int` 4 bytes (32 bits, de –2.147.483.648 a 2.147.483.647);
- `long` 8 bytes (64 bits, de –9.223.372.036.854.775.808 a 9.223.372.036.854.775.807).

Decorar o tamanho dos primitivos para prova

Não há a necessidade de decorar o intervalo e tamanho de todos os tipos de primitivos para a prova. O único intervalo cobrado é o do `byte` (-127 a 128).

É importante também saber que o `char`, apesar de ter o mesmo tamanho de um `short`, não consegue armazenar todos os números que cabem em um `short`, já que o `char` só armazena números positivos.

Para saber mais: calculando o intervalo de valores

Dado o número de bits **N** do tipo primitivo inteiro, para saber os valores que ele aceita usamos a seguinte conta:

```
-2^(n-1) a 2^(n-1) -1
```

O `char`, por ser apenas positivo, tem intervalo:

```
0 a 2^(16) -1
```

Os tipos ponto flutuante têm os seguintes tamanhos em notação científica:

- `float` 4 bytes (32 bits, de +/–1.4 * 10 ^45 a +/–3.4028235 * 10 ^38);
- `double` 8 bytes (64 bits, de +/–4.9 * 10 ^324 a +/–1.7976931348623157 * 10 ^308).

Todos os números de ponto flutuante também podem assumir os seguintes valores:

- +/– infinity
- +/- 0
- NaN (Not a Number)

Literais

Na codificação, muitas vezes o programador coloca os valores das variáveis diretamente no código-fonte. Quando isso ocorre, dizemos que o valor foi literalmente escrito no código, ou seja, é um **valor literal**.

Todos os valores primitivos maiores que `int` podem ser expressos literalmente. Por outro lado, as referências (valores não primitivos) não podem ser expressas de maneira literal (não conseguimos colocar direto os endereços de memória dos objetos).

Ao inicializar uma variável, podemos explicitar que queremos que ela seja do tipo `double` ou `long` usando a letra específica:

```
// compila pois 737821237891232 é um double válido
System.out.println(737821237891232d);

// compila pois 737821237891232 é um long válido
System.out.println(737821237891232l);

// nao compila pois 737821237891232 é um valor maior que
// o int aceita
System.out.println(737821237891232);
```

Da mesma maneira, o compilador é um pouco esperto e percebe se você tenta quebrar o limite de um `int` muito facilmente:

```
// compila pois 737821237891232l é um long válido
long l = 737821237891232l;

// não compila pois o compilador não é bobo assim
int i = l;

// booleanos
System.out.println(true); // booleano verdadeiro
System.out.println(false); // booleano falso

// números simples são considerados inteiros
System.out.println(1); // int

// números com casa decimal são considerados double.
// Também podemos colocar uma letra "D" ou "d" no final
System.out.println(1.0); //double
System.out.println(1.0D); //double

// números inteiros com a letra "L" ou "l"
// no final são considerados long.
System.out.println(1L); //long

// números com casa decimal com a letra "F" ou "f"
// no final são considerados float.
System.out.println(1.0F); //float
```

Bases diferentes

No caso dos números inteiros, podemos declarar usando bases diferentes. O Java suporta a base **decimal** e mais as bases **octal**, **hexadecimal** e **binária**.

Um número na base octal tem que começar com um zero à esquerda e pode usar apenas os algarismos de 0 a 7:

```
int i = 0761; // base octal

System.out.println(i); // saída: 497
```

E na hexadecimal, começa com `0x` ou `0X` e usa os algarismos de 0 a 15. Como não existe um algarismo "15", usamos letras para representar algarismos de "10" a "15", no caso, "A" a "F", maiúsculas ou minúsculas:

```
int j = 0xAB3400; // base hexadecimal
System.out.println(j); // saída: 11219968
```

Já na base binária, começamos com `0b`, e só podemos usar "0" e "1":

```
int b = 0b100001011; // base binária
System.out.println(b); // saída: 267
```

Não é necessário aprender a fazer a conversão entre as diferentes bases e a decimal. Apenas saber quais são os valores possíveis em cada base, para identificar erros de compilação como o que segue:

```
int i = 0769; // erro, base octal não pode usar 9
```

Notação científica

Ao declarar `doubles` ou `floats`, podemos usar a notação científica:

```
double d = 3.1E2;
System.out.println(d); // 310.0

float e = 2e3f;
System.out.println(e); // 2000.0

float f = 1E4F;
System.out.println(f); // 10000.0
```

Usando underlines em literais

A partir do Java 7, existe a possibilidade de usarmos *underlines* (_) quando estamos declarando literais para facilitar a leitura do código:

```
int a = 123_456_789;
```

Existem algumas regras sobre onde esses *underlines* podem ser posicionados nos literais, e caso sejam colocados em locais errados resultam em erros de compilação. A regra básica é que eles só podem ser posicionados com **valores numéricos em ambos os lados**. Vamos ver alguns exemplos:

```
int v1 = 0_100_267_760;        // ok
int v2 = 0_x_4_13;             // erro, _ antes e depois do x
int v3 = 0b_x10_BA_75;         // erro, _ depois do b
int v4 = 0b_10000_10_11;     // erro, _ depois do b
int v5 = 0xa10_AF_75;         // ok, apesar de ser letra
                            // representa dígito
int v6 = _123_341;            // erro, inicia com _
int v7 = 123_432_;            // erro, termina com _
int v8 = 0x1_0A0_11;        // ok
int v9 = 144__21_12;        // ok
```

A mesma regra se aplica a números de ponto flutuante:

```
double d1 = 345.45_e3;        // erro, _ antes do e
double d2 = 345.45e_3;        // erro, _ depois do e
double d3 = 345.4_5e3;        // ok
double d4 = 34_5.45e3_2;    // ok
double d5 = 3_4_5.4_5e3;    // ok
double d6 = 345._45F;        // erro, _ depois do .
double d7 = 345_.45;        // erro, _ antes do .
double d8 = 345.45_F;        // erro, _ antes do indicador de
                            // float
double d9 = 345.45_d;        // erro, _ antes do indicador de
                            // double
```

Iniciando chars

Os chars são iniciados colocando o caractere desejado entre **aspas simples**:

```
char c = 'A';
```

Mas podemos iniciar com números também. Neste caso, o número representa a posição do caractere na tabela unicode:

```
char c = 65;
System.out.println(c); // imprime A
```

Não é necessário decorar a tabela unicode, mas é preciso prestar atenção a pegadinhas como a seguinte:

```
char sete = 7; // número, pois não está entre aspas simples
System.out.println(sete);  // Não imprime nada!!!!
```

Quando usando programas em outras línguas, às vezes queremos usar caracteres unicode, mas não temos um teclado com tais teclas (árabe, chinês etc.). Neste caso, podemos usar uma representação literal de um caractere unicode em nosso código, iniciando o `char` com `\u`:

```
char c = '\u03A9'; // unicode
System.out.println(c); // imprime a letra grega ômega
```

Identificadores

Quando escrevemos nossos programas, usamos basicamente dois tipos de termos para compor nosso código: identificadores e palavras reservadas.

Chamamos de **identificadores** as palavras definidas pelo programador para nomear variáveis, métodos, construtores, classes, interfaces etc.

Já **palavras reservadas** ou **palavras-chave** são termos predefinidos da linguagem que podemos usar para definir comandos (`if`, `for`, `class`, entre outras).

São diversas palavras-chave na linguagem java:

- abstract

- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- false
- final
- finally
- float
- for
- goto

- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- null
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this

- throw
- throws
- transient
- true
- try
- void
- volatile
- while

NULL, FALSE E TRUE

Outras três palavras reservadas que não aparecem nessa lista são `true`, `false` e `null`. Mas, segundo a especificação na linguagem Java, esses três termos são considerados *literais* e não palavras-chave (embora também sejam reservadas), totalizando 53 palavras reservadas.

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

Identificadores válidos devem seguir as seguintes regras:

- Não podem ser igual a uma palavra-chave;
- Podem usar letras (unicode), números, `$` e _;
- O primeiro caractere **não** pode ser um número;
- Podem possuir qualquer número de caracteres.

Os identificadores são *case sensitive*, ou seja, respeitam maiúsculas e minúsculas:

```
int umNome; // ok
int umnome;    // ok, diferente do anterior
int _num;    // ok
int $_ab_c; // ok
int x_y;    // ok
int false;    // inválido, palavra reservada
int x-y;     // inválido, traço
int 4num;     // inválido, começa com número
int av#f;     // inválido, #
int num.spc; // inválido, ponto no meio
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int
4         idade
5         = 100;
6         System.out.println(idade);
7     }
8 }
```

a) O código não compila: erros a partir da linha que define uma variável do tipo `int`.
b) O código não compila: a variável `idade` não foi inicializada, mas foi usada em `System.out.println`.
c) O código compila e imprime 0.
d) O código compila e imprime 100.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int idade;
4         if(args.length > 0) {
5             idade = Integer.parseInt(args[0]);
6         } else {
```

```
7                 System.err.println("Por favor passe sua idade como
8                                   primeiro parâmetro");
9             }
10            System.out.println("Sua idade é " + idade);
11        }
12 }
```

a) Não compila: erro na linha que tenta acessar a variável `idade`.

b) Compila e imprime 0 ou a idade que for passada na linha de comando.

c) Compila e imprime a idade que for passada na linha de comando.

d) Compila e imprime a mensagem de erro ou "Sua idade é "e a idade.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         boolean array = new boolean[300];
4         System.out.println(array[3]);
5     }
6 }
```

a) Imprime `true`.

b) Imprime `false`.

c) Imprime `0`.

d) Imprime `-1`.

e) Imprime `null`.

f) Não compila.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         boolean[] array = new boolean[300];
4         System.out.println(array[3]);
5     }
6 }
```

a) Imprime `true`.

b) Imprime `false`.

c) Imprime `0`.

d) Imprime `-1`.

e) Imprime `null`.

f) Não compila.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         boolean argumentos;
4         if(args.length > 0)
5             argumentos = 1;
6         else
7             argumentos = 0;
8         System.out.println(argumentos);
9     }
10 }
```

a) Não compila: o método de impressão não recebe `boolean`.

b) Não compila: atribuição inválida.

c) Não compila: o método `length` de array não é uma propriedade.

d) Não compila: o método `length` de `String[]` não é uma propriedade.

e) Compila e imprime `0` ou `1`.

f) Compila e imprime `false` ou `true`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int n = 09;
4         int m = 03;
```

```
5             int x = 1_000;
6             System.out.println(x - n + m);
7         }
8 }
```

a) Não compila: erro na linha que declara `n`.

b) Não compila: erro na linha que declara `x`.

c) Não compila: erro na linha que declara `m`.

d) Compila e imprime um número menor que 1000.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         for(char c='a';c <= 'z';c++) {
4             System.out.println(c);
5         }
6     }
7 }
```

a) Não compila: não podemos somar um em um caractere.

b) Não compila: não podemos comparar caracteres com `<`.

c) Compila e imprime o alfabeto entre a e z, inclusive.

8) Qual das palavras a seguir não é reservada em Java?

a) strictfp

b) native

c) volatile

d) transient

e) instanceOf

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         boolean BOOLEAN = false;
4         if(BOOLEAN) {
5             System.out.println("Sim");
6         }
7     }
8 }
```

a) Não compila: não podemos declarar uma variável com o nome de uma palavra reservada.

b) Não compila: não podemos declarar uma variável iniciando com letras maiúsculas.

c) Compila e roda, imprimindo `Sim`.

d) Compila e roda, não imprimindo nada.

4.2 Diferencia entre variáveis de referências a objetos e tipos primitivos

As variáveis de tipos primitivos de fato armazenam os valores (e não ponteiros/referências). Ao se atribuir o valor de uma variável primitiva a uma outra variável, o valor é copiado, e o original não é alterado:

```
int a = 10;
int b = a; // copiando o valor de a para b
b++; // somando 1 em b
System.out.println(a); // continua com 10.
```

Os programas construídos com o modelo orientado a objetos utilizam, evidentemente, objetos. Para acessar um atributo ou invocar um método de qualquer objeto, é necessário que tenhamos armazenada uma **referência** para o mesmo.

Uma variável de referência é um ponteiro para o endereço de memória onde o objeto se encontra. Ao atribuirmos uma variável de referência a outra, estamos copiando a referência, ou seja, fazendo com que as duas variáveis apontem para o mesmo objeto, e não criando um novo objeto:

```
class Objeto {
    int valor;
}

class Teste{
    public static void main(String[] args){
        Objeto a = new Objeto();
        Objeto b = a; // agora b aponta para o mesmo objeto de a

        a.valor = 5;

        System.out.println(b.valor); // imprime 5
    }
}
```

Duas referências são consideradas iguais somente se elas estão apontando para o mesmo objeto. Mesmo que os objetos que elas apontem sejam iguais, ainda são referências para objetos diferentes:

```
Objeto a = new Objeto();
a.valor = 5;

Objeto b = new Objeto();
b.valor = 5;

Objeto c = a;

System.out.println(a == b); // false
System.out.println(a == c); // true
```

Veremos bastante sobre comparação de tipos primitivos e de referências mais à frente.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int x = 15;
4         int y = x;
```

```
5         y++;
6         x++;
7         int z = y;
8         z--;
9         System.out.println(x + y + z);
10    }
11 }
```

a) Imprime 43.

b) Imprime 44.

c) Imprime 45.

d) Imprime 46.

e) Imprime 47.

f) Imprime 48.

g) Imprime 49.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     int v = 15;
3 }
4 class A {
5     public static void main(String[] args) {
6         B x = new B();
7         B y = x;
8         y.v++;
9         x.v++;
10        B z = y;
11        z.v--;
12        System.out.println(x.v + y.v + z.v);
13    }
14 }
```

a) Imprime 43.

b) Imprime 44.

c) Imprime 45.

d) Imprime 46.

e) Imprime 47.

f) Imprime 48.

g) Imprime 49.

4.3 Leia ou escreva para campos de objetos

Ler e escrever propriedades em objetos é uma das tarefas mais comuns em um programa java. Para acessar um atributo, usamos o operador `.` (ponto), junto a uma variável de referência para um objeto. Veja a seguinte classe:

```
class Carro {
    String modelo;
    int ano;

    public Carro() { ano = 2014; }

    public String getDadosDeImpressao() {
        return modelo + " - " + ano;
    }

    public void setModelo(String m) {
        this.modelo = m;
    }
}
```

Vamos escrever um código para usar essa classe:

```
1 Carro a = new Carro();
2 a.modelo = "Palio";      // acessando diretamente o atributo
3 a.setModelo("Palio");     // acessando o atributo por um método
4
5 // acessando o método e passando o retorno como argumento para
6 // o método println
7 System.out.println(a.getDadosDeImpressao());
8
9 }
```

As linhas 2 e 3 têm exatamente o mesmo efeito. Como iniciamos o valor da propriedade `ano` no construtor, ao chamar o método `imprimeDados`, o valor `2014` é exibido junto ao nome do modelo.

Quando estamos dentro da classe, não precisamos de nenhum operador para acessar os atributos de instância da classe. Opcionalmente, podemos usar a palavra-chave `this`, que serve como uma variável de referência para o próprio objeto onde o código está sendo executado:

```
class Carro{
    int ano;
    int modelo;

    public Carro(){
        modelo = "Indefinido";     // acessando variável de
                                  // instancia sem o this
        this.ano = 2014;          // acessando com o this.
    }
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B{
2      int c;
3      void c(int c) {
4          c = c;
5      }
6  }
7  class A {
8      public static void main(String[] args) {
9          B b = new B();
10         b.c = 10;
11         System.out.println(b.c);
12         b.c(30);
13         System.out.println(b.c);
14     }
15 }
```

a) Não compila: conflito de nome de variável membro e método em B.

b) Não compila: conflito de nome de variável membro e variável local em B.
c) Compila e roda, imprimindo 10 e 30.
d) Compila e roda, imprimindo outro resultado.

4.4 Explique o ciclo de vida de um objeto (criação, "dereferência" e garbage collection)

O ciclo de vida dos objetos java está dividido em três fases distintas. Vamos conhecê-las e entender o que cada uma significa.

Criação de objetos

Toda vez que usamos o operador `new`, estamos criando uma nova instância de um objeto na memória:

```
class Pessoa {
    String nome;
}

class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa(); // criando um novo objeto do
                                 // tipo Pessoa
    }
}
```

Repare que há uma grande diferença entre criar um objeto e declarar uma variável. A variável é apenas uma referência, um ponteiro, não contém um objeto de verdade.

```
// Apenas declarando a variável,
// nenhum objeto foi criado aqui
Pessoa p;

// Agora um objeto foi criado e atribuído a variável
p = new Pessoa();
```

Objeto acessível

A partir do momento em que um objeto foi criado e atribuído a uma variável, dizemos que o objeto está **acessível**, ou seja, podemos usá-lo em nosso programa:

```
Pessoa p = new Pessoa(); // criação
p.nome = "Mário"; // acessando e usando o objeto
```

Objeto inacessível

Um objeto é acessível enquanto for possível "alcançá-lo" através de alguma referência direta ou indireta. Caso não exista nenhum caminho direto ou indireto para acessar esse objeto, ele se torna **inacessível**.:

```
1 Pessoa p = new Pessoa();
2 p.nome = "Mário";
3
4 // atribuímos a p o valor null
5 // o objeto não está mais acessível
6 p = null
7
8 // criando um objeto sem variável
9 new Pessoa();
```

Nesse código, criamos um objeto do tipo `Pessoa` e o atribuímos à variável `p`. Na linha **6** atribuímos `null` a `p`. O que acontece com o objeto anterior? Ele simplesmente não pode mais ser acessado por nosso programa, pois não temos nenhum ponteiro para ele. O mesmo pode ser dito do objeto criado na linha **9**. Após essa linha, não conseguimos mais acessar esse objeto.

Outra maneira de ter um objeto inacessível é quando o escopo da variável que aponta para ele termina:

```
int valor = 100;
if( valor > 50) {
    Pessoa p = new Pessoa();
    p.nome = "João";
} // Após esta linha, o objeto do tipo Pessoa não está mais
  // acessível
```

Garbage Collector

Todo objeto inacessível é considerado elegível para o *garbage collector*. Algumas questões da prova perguntam quantos objetos são elegíveis ao garbage collector ao final de algum trecho de código:

```
public class Bla {
     int b;
     public static void main(String[] args) {
         Bla b;
        for (int i = 0; i < 10; i++) {
             b = new Bla();
             b.b = 10;
         }
           System.out.println("fim");
     }
 }
```

Ao chegar na linha 9, temos **9** objetos elegíveis para o Garbage Collector.

> ### Objetos elegíveis X Objetos coletados
>
> O *garbage collector* roda em segundo plano juntamente com sua aplicação java. Não é possível prever quando ele será executado, portanto não se pode dizer com certeza quantos objetos foram efetivamente coletados em um certo ponto da aplicação. O que podemos determinar é quantos objetos são elegíveis para a coleta. A prova pode tentar se aproveitar do descuido do desenvolvedor aqui: nunca temos certeza de quantos objetos passaram pelo garbage collector, logo, somente indique quantos estão passíveis de serem coletados.

Por fim, é importante ver um exemplo de referência indireta, no qual nenhum objeto pode ser "garbage coletado":

```
1 import java.util.*;
2 class Carro {
3
```

```
}
class Carros {
    List<Carro> carros = new ArrayList<Carro>();
}
class Teste {
    public static void main(String args[]) {
        Carros carros = new Carros();
        for(int i = 0; i < 100; i++)
            carros.carros.add(new Carro());
        // até essa linha todos ainda podem ser alcançados
    }
}
```

Nesse código, por mais que tenhamos criados 100 carros e um objeto do tipo `Carros`, nenhum deles pode ser garbage coletado pois todos podem ser alcançados direta ou indiretamente através de nossa *thread* principal.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B b;
        for(int i = 0;i < 10;i++)
            b = new B();
        System.out.println("Finalizando!");
    }
}
```

a) Não compila.

b) Compila e garbage coleta 10 objetos do tipo `B` na linha do `System.out`.

c) Compila e não podemos falar quantos objetos do tipo `B` foram garbage coletados na linha do `System.out`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B b = new B();
        for(int i = 0;i < 10;i++)
            b = new B();
        System.out.println("Finalizando!");
    }
}
```

a) Não compila.

b) Compila e 10 objetos do tipo `B` podem ser garbage coletados ao chegar na linha do `System.out`.

c) Compila e 11 objetos do tipo `B` podem ser garbage coletados ao chegar na linha do `System.out`.

d) Compila e garbage coleta 11 objetos do tipo `B` na linha do `System.out`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B[] bs = new B[100];
        System.out.println("Finalizando!");
    }
}
```

a) Compila e 100 objetos do tipo `B` são criados, mas não podemos falar nada sobre o garbage collector ter jogado os objetos fora na linha do `System.out`.

b) Compila e nenhum objeto do tipo `B` é criado.

c) Compila, cria 100 e joga fora todos os objetos do tipo `B` ao chegar no `System.out`.

4.5 Chame métodos em objetos

Além de acessar atributos, também podemos invocar métodos em um objeto. Para isso usamos o operador `.` (ponto), junto a uma variável de referência para um objeto. Deve-se prestar atenção ao número e tipo de parâmetros do método, além do seu retorno. Métodos declarados como `void` não possuem retorno, logo, não podem ser atribuídos a nenhuma variável ou passado para outro método como parâmetro:

```
class Pessoa{

    String nome;

    public String getNome(){
        return nome;
    }

    public void setNome(String nome){
        this.nome = nome;
    }
}

class Teste{
    public static void main(String[] args){
        Pessoa p = new Pessoa();

        //chamando método na variável de ref.
        p.setNome("Mario");

        //Atribuindo o retorno do método a variável.
        String nome = p.getNome();

        // erro, método é void
        String a = p.setNome("X");
```

```
    }
}
```

Quando um método está sendo invocado em um objeto, podemos chamar outro método no mesmo objeto através da invocação direta ao nome do método:

```
class A {
    void metodo1() {
        metodo2(); // chama o metodo2 no objeto onde metodo1 foi
                   // chamado
    }
    void metodo2() {
    }
}
```

Argumentos variáveis: varargs

A partir do Java 5, **varargs** possibilitam um método que receba um número variável (não fixo) de parâmetros. É a maneira de receber um array de objetos e possibilitar uma chamada mais fácil do método.

Um caso especial é quando método recebe um argumento variável (`varargs`). Neste caso, podemos chamá-lo com qualquer número de argumentos:

```
class Calculadora{
    public int soma(int... nums){
        int total = 0;
        for (int a : nums){
            total+= a;
        }
        return total;
    }

}
```

`nums` realmente é um array aqui, você pode fazer um `for` usando o `length`, ou mesmo usar o `enhanced for`. A invocação desse método pode ser feita de várias maneiras:

```
public static void main (String[] args){
    Calculadora c = new Calculadora();

    //Todas as chamadas abaixo sao válidas
    System.out.println(c.soma());
    System.out.println(c.soma(1));
    System.out.println(c.soma(1,2));
    System.out.println(c.soma(1,2,3,4,5,6,7,8,9));
}
```

Em todos os casos, um array será criado, nunca `null` será passado. Um parâmetro `varargs` deve ser sempre o último da assinatura do método para evitar ambiguidade. Isso implica que apenas um dos parâmetros de um método seja `varargs`. E repare que os argumentos variáveis têm que ser do mesmo tipo.

E será dada a prioridade para o método que já podia existir antes no Java 1.4:

```
void metodo(int ... x) { }
void metodo(int x) {}

metodo(5);
```

Isso vai invocar o segundo método. Podemos também passar um array de `ints` para um método que recebe um `varargs`:

```
void metodo(int ... x) { }

metodo(new int[] {1,2,3,4});
```

Mas nunca podemos chamar um método que recebe array como se ele fosse `varargs`:

```
void metodo(int[] x) { }

metodo(1,2,3); // não compila
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     void x() {
3         System.out.println("vazio");
4     }
5     void x(String... args) {
6         System.out.println(args.length);
7     }
8 }
9 class C {
10     void x(String... args) {
11         System.out.println(args.length);
12     }
13     void x() {
14         System.out.println("vazio");
15     }
16 }
17 class A {
18     public static void main(String[] args) {
19         new B().x();
20         new C().x();
21     }
22 }
```

a) Não compila: conflito entre método com `varargs` e sem argumentos.

b) Compila e imprime vazio/vazio.

c) Compila e imprime vazio/0.

d) Compila e imprime 0/vazio.

e) Compila e imprime 0/0.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     void x(int... x) {
3         System.out.println(x.length);
4     }
5 }
6 class A {
```

```
7       public static void main(String[] args) {
8           new B().x(23789,673482);
9       }
10  }
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Compila e ao rodar imprime os dois números.

c) Compila e ao rodar imprime 2.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1   class B{
2       void x(int... x) {
3           System.out.println(x.length);
4       }
5   }
6   class A {
7       public static void main(String[] args) {
8           new B().x(new int[]{23789,673482});
9       }
10  }
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Não compila: não podemos passar um array para um `varargs`.

c) Compila e ao rodar imprime os dois números.

d) Compila e ao rodar imprime 1.

e) Compila e ao rodar imprime 2.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1   class B{
2       void x(Object... x) {
3           System.out.println(x.length);
4       }
5   }
6   class A {
```

```
7      public static void main(String[] args) {
8          new B().x(new Object[]{23789,673482});
9      }
10 }
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Não compila: não podemos passar um array para um `varargs`.

c) Compila e ao rodar imprime os dois números.

d) Compila e ao rodar imprime 1.

e) Compila e ao rodar imprime 2.

4.6 Manipule dados usando a classe StringBuilder e seus métodos

Para suportar Strings mutáveis, o Java possui as classes `StringBuffer` e `StringBuilder`. A operação mais básica é o `append` que permite concatenar ao mesmo objeto:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum");
sb.append(" - ");
sb.append("Ensino e Inovação");

System.out.println(sb);  // Caelum - Ensino e Inovação
```

Repara que o `append` não devolve novos objetos como em `String`, mas altera o próprio `StringBuffer`, que é mutável.

Podemos criar um objeto desse tipo de diversas maneiras diferentes:

```
// vazio
StringBuilder sb1 = new StringBuilder();
// conteudo inicial
StringBuilder sb2 = new StringBuilder("java");
// tamanho inicial do array para colocar a string
StringBuilder sb3 = new StringBuilder(50);
// baseado em outro objeto do mesmo tipo
StringBuilder sb4 = new StringBuilder(sb2);
```

Tenha cuidado: ao definir o tamanho do array, não estamos criando uma `String` de tamanho definido, somente um array desse tamanho que será utilizado pelo `StringBuilder`, portanto:

```
StringBuilder sb3 = new StringBuilder(50);
System.out.println(sb3); // linha em branco
System.out.println(sb3.length()); // 0
```

As classes `StringBuffer` e `StringBuilder` têm exatamente a mesma interface (mesmos métodos), sendo que a primeira é *thread-safe* e a última não (e foi adicionada no Java 5). Quando não há compartilhamento entre threads, use sempre que possível a `StringBuilder`, que é mais rápida por não precisar se preocupar com *locks*.

Inclusive, em Java, quando fazemos concatenação de Strings usando o `+`, por baixo dos panos, é usado um `StringBuilder`. Não existe a operação `+` na classe `String`. O compilador troca todas as chamadas de concatenação por `StringBuilders` (podemos ver isso no bytecode compilado).

Principais métodos de StringBuffer e StringBuilder

Há a família de métodos `append` com overloads para receber cada um dos primitivos, Strings, arrays de chars, outros `StringBuffer` etc. Todos eles devolvem o próprio `StringBuffer/Builder` o que permite chamadas encadeadas:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum").append(" - ").append("Ensino e Inovação");
System.out.println(sb);  // Caelum - Ensino e Inovação
```

O método `append` possui uma versão que recebe `Object` e chama o método `toString` de seu objeto.

Há ainda os métodos `insert` para inserir coisas no meio. Há versões que recebem primitivos, Strings, arrays de char etc. Mas todos têm o primeiro argumento recebendo o índice onde queremos inserir:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum - Inovação");
sb.insert(9, "Ensino e ");
```

```
System.out.println(sb); // Caelum - Ensino e Inovação
```

Outro método que modifica é o `delete`, que recebe os índices inicial e final:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum - Ensino e Inovação");
sb.delete(6, 15);

System.out.println(sb); // Caelum e Inovação
```

Para converter um `StringBuffer/ Builder` em `String`, basta chamar o `toString` mesmo. O método `reverse` inverte seu conteúdo:

```
System.out.println(new StringBuffer("guilherme").reverse());
                                                // emrehliug
```

Fora esses, também há o `trim`, `charAt`, `length()`, `equals`, `indexOf`, `lastIndexOf`, `substring`.

Cuidado, pois o método `substring` não altera o valor do seu `StringBuilder` ou `StringBuffer`, mas retorna a `String` que você deseja. Existe também o método `subSequence` que recebe o início e o fim e funciona da mesma maneira que o `substring` com dois argumentos.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         StringBuilder sb = new StringBuilder();
4         sb.append("guilherme").delete(2,3);
5         System.out.println(sb);
6     }
7 }
```

a) O código não compila: erro na linha que tenta imprimir o `StringBuilder`.

b) O código compila e imprime `glherme`.

c) O código compila e imprime `guherme`.
d) O código compila e imprime `gilherme`.
e) O código compila e imprime `gulherme`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3       StringBuilder sb = new StringBuilder("guilherme");
4       System.out.println(sb.indexOf("e") + sb.lastIndexOf("e"));
5       System.out.println(sb.indexOf("k") + sb.lastIndexOf("k"));
6     }
7 }
```

a) O código imprime 13 e -2.
b) O código imprime 13 e 0.
c) O código imprime 13 e -1.
d) O código imprime 13 e 8.
e) O código imprime 13 e 10.
f) O código imprime 15 e -2.
g) O código imprime 15 e 0.
h) O código imprime 15 e -1.
i) O código imprime 15 e 8.
j) O código imprime 15 e 10.

4.7 Criando e manipulando Strings

Existem duas maneiras tradicionais de criar uma `String`:

```
String nomeDireto = "Java";
String nomeIndireto = new String("Java");
```

A comparação entre esses dois tipos de criação de `Strings` é feita na seção *Test equality between strings and other objects using == and equals()* 5.3

Existem outras maneiras não tão comuns:

```
char[] nome = new char[]{'J', 'a', 'v', 'a'};
String nomeComArray = new String(nome);

StringBuilder sb1 = new StringBuilder("Java");
String nome1 = new String(sb1);

StringBuffer sb2 = new StringBuffer("Java");
String nome2 = new String(sb2);
```

Como uma `String` não é um tipo primitivo, ela pode ter valor `null`, lembre-se disso:

```
String nome = null; // null explicito
```

Podemos concatenar `Strings` com o `+`:

```
String nome = "Certificação" + " " + "Java";
```

Caso tente concatenar `null` com uma `String`, temos a conversão de `null` para `String`:

```
String nula = null;
System.out.println("nula: " + nula); // imprime nula: null
```

O Java faz a conversão de tipos primitivos para `Strings` automaticamente, mas lembre-se da precedência de operadores:

```
String nome = "Certificação" + ' ' + "Java" + ' ' + 1500;
System.out.println(nome);

String nome2 = "Certificação";
nome2 += ' ' + "Java" + ' ' + 1500;
System.out.println(nome2);

String valor = 15 + 00 + " certificação";
System.out.println(valor); // imprime "15 certificação",
                           // primeiro efetuando uma soma
```

Strings são imutáveis

O principal ponto sobre Strings é que elas são imutáveis:

```
String s = "caelum";
s.toUpperCase();
System.out.println(s);
```

Esse código imprime `caelum` em minúscula. Isso porque o método `toUpperCase` não altera a `String` original. Na verdade, se olharmos o javadoc da classe `String` vamos perceber que **todos os métodos que parecem modificar uma String na verdade devolvem uma nova**.

```
String s = "caelum";
String s2 = s.toUpperCase();
System.out.println(s2);
```

Agora sim imprimirá `CAELUM`, uma nova String. Ou, usando a mesma *referência*:

```
String s = "caelum";
s = s.toUpperCase();
System.out.println(s);
```

Para tratarmos de "strings mutáveis", usamos as classes `StringBuffer` e `StringBuilder`.

Lembre-se que a `String` possui um array por trás e, seguindo o padrão do Java, suas posições começam em 0:

```
// 0=g, devolve 'g'
char caracter0 = "guilherme".charAt(0);

// 0=g 1=u, devolve 'u'
char caracter1 = "guilherme".charAt(1);

// 0=g 1=u 2=i, devolve 'i'
char caracter2 = "guilherme".charAt(2);
```

Cuidado ao acessar uma posição indevida, você pode levar um `StringIndexOutOfBoundsException` (atenção ao nome da `Exception`, não é `ArrayIndexOutofBoundsException`):

```
char caracter20 = "guilherme".charAt(20); // exception
char caracterMenosUm = "guilherme".charAt(-1); // exception
```

Principais métodos de String

O método `length` imprime o tamanho da `String`:

```
String s = "Java";
System.out.println(s.length()); // 4
System.out.println(s.length); // não compila: não é atributo
System.out.println(s.size()); // não compila: não existe size
                              // em String Java
```

Já o método `isEmpty` diz se a `String` tem tamanho zero:

```
System.out.println("".isEmpty()); // true
System.out.println("java".isEmpty()); // false
System.out.println(" ".isEmpty()); // false
```

Devolvem uma nova String:

- `String toUpperCase()` tudo em maiúscula;
- `String toLowerCase()` tudo em minúsculo;
- `String trim()` retira espaços em branco no começo e no fim;
- `String substring(int beginIndex, int endIndex)` devolve a substring a partir dos índices de começo e fim;
- `String substring(int beginIndex)` semelhante ao anterior, mas toma a substring a partir do índice passado até o final da String;
- `String concat(String)` concatena o parâmetro ao fim da String atual e devolve o resultado;
- `String replace(char oldChar, char newChar)` substitui todas as ocorrências de determinado `char` por outro;
- `String replace(CharSequence target, CharSequence replacement)` substitui todas as ocorrências de determinada `CharSequence` (como String) por outra.

O método `trim` limpa caracteres em branco nas duas pontas da `String`:

```
System.out.println("     ".trim()); // imprime só a quebra de
                                    // linha do println
System.out.println(" ".trim().isEmpty()); // true
System.out.println(" guilherme "); // imprime 'guilherme'
System.out.println(". ."); // imprime '. .'
```

O método `replace` substituirá todas as ocorrências de um texto por outro:

```
System.out.println("java".replace("j", "J")); // Java
System.out.println("guilherme".replace("e", "i")); // guilhirmi
```

Podemos sempre fazer o *chaining* e criar uma sequência de "transformações" que retornam uma nova `String`:

```
String parseado = "  Quero tirar um certificado oficial de
                     Java!  ".toUpperCase().trim();

// imprime: "QUERO TIRAR UM CERTIFICADO OFICIAL DE JAVA!"
System.out.println(parseado);
```

Para extrair pedaços de uma `String`, usamos o método `substring`. Cuidado ao usar o método `substring` com valores inválidos, pois eles jogam uma `Exception`. O segredo do método `susbtring` é que ele não inclui o caractere da posição final, mas inclui o caractere da posição inicial:

```
String texto = "Java";

// ava
System.out.println(texto.substring(1));

// StringIndexOutOfBoundsException
System.out.println(texto.substring(-1));

// StringIndexOutOfBoundsException
System.out.println(texto.substring(5));

// Java
```

```
System.out.println(texto.substring(0, 4));

// ava
System.out.println(texto.substring(1, 4));

// Jav
System.out.println(texto.substring(0, 3));

// StringIndexOutOfBoundsException
System.out.println(texto.substring(0, 5));

// StringIndexOutOfBoundsException
System.out.println(texto.substring(-1, 4));
```

Comparação:

- `boolean equals(Object)` compara igualdade caractere a caractere (herdado de `Object`);
- `boolean equalsIgnoreCase(String)` compara caractere a caractere ignorando maiúsculas/minúsculas;
- `int compareTo(String)` compara as 2 Strings por ordem lexicográfica (vem de `Comparable`);
- `int compareToIgnoreCase(String)` compara as 2 Strings por ordem lexicográfica ignorando maiúsculas/minúsculas.

E aqui, todas as variações desses métodos. Não precisa saber o número exato que o `compareTo` retorna, basta saber que será negativo caso a `String` na qual o método for invocado vier antes, zero se for igual, positivo se vier depois do parâmetro passado:

```
String texto = "Certificado";
System.out.println(texto.equals("Certificado")); // true
System.out.println(texto.equals("certificado")); // false
System.out.println(texto.equalsIgnoreCase("certificado"));//true

System.out.println(texto.compareTo("Arnaldo")); // 2
```

```
System.out.println(texto.compareTo("Certificado")); // 0
System.out.println(texto.compareTo("Grécia")); // -4

System.out.println(texto.compareTo("certificado")); // -32

System.out.println(texto.compareToIgnoreCase("certificado"));//0
```

Buscas simples:

- `boolean contains(CharSequence)` devolve `true` se a String contém a sequência de `chars`;
- `boolean startsWith(String)` devolve `true` se começa com a String do parâmetro;
- `boolean endsWith(String)` devolve `true` se termina com a String do parâmetro;
- `int indexOf(char)` e `int indexOf(String)` devolve o índice da primeira ocorrência do parâmetro;
- `int lastIndexOf(char)` e `int lastIndexOf(String)` devolve o índice da última ocorrência do parâmetro.

O código a seguir exemplifica todos os casos desses métodos:

```
String texto = "Pretendo fazer a prova de certificação de Java";

System.out.println(texto.indexOf("Pretendo")); // imprime 0
System.out.println(texto.indexOf("Pretendia")); // imprime -1
System.out.println(texto.indexOf("tendo")); // imprime 3

System.out.println(texto.indexOf("a")); // imprime 10
System.out.println(texto.lastIndexOf("a")); // imprime 45
System.out.println(texto.lastIndexOf("Pretendia")); //imprime -1

System.out.println(texto.startsWith("Pretendo")); // true
System.out.println(texto.startsWith("Pretendia")); // false

System.out.println(texto.endsWith("Java")); // true
System.out.println(texto.endsWith("Oracle")); // false
```

1) Considere o seguinte código dentro de um `main`:

```
class A{
    public static void main(String [] args){
        String s = "aba";
        for(int i = 0; i < 9; i++) {
            s = s +"aba";
        }
        System.out.println(s.length);
    }
}
```

a) Não compila.

b) Compila e imprime 3.

c) Compila e imprime 30.

d) Compila e imprime 33.

e) Compila e imprime 36.

2) Dada a seguinte classe:

```
class B {
    String msg;

    void imprime() {
        if (!msg.isEmpty())
            System.out.println(msg);
        else
            System.out.println("vazio");
    }
}
```

O que acontece se chamarmos `new B().imprime()`?

a) Não compila.

b) Compila, mas dá exceção na hora de rodar.

c) Compila, roda e não imprime nada.

d) Compila, roda e imprime "vazio".

3) Dada a seguinte classe:

```
class B {

    void imprime() {
        String msg;
        if (!msg.isEmpty())
            System.out.println(msg);
        else
            System.out.println("vazio");
    }
}
```

O que acontece se chamarmos `new B().imprime()` ?

a) Não compila.

b) Compila, mas dá exceção na hora de rodar.

c) Compila, roda e não imprime nada.

d) Compila, roda e imprime "vazio".

4) Qual é a saída nos dois casos?

```
String s = "Caelum";
s.concat(" - Ensino e Inovação");
System.out.println(s);

StringBuffer s = new StringBuffer("Caelum");
s.append(" - Ensino e Inovação");
System.out.println(s);
```

a) 'Caelum' e 'Caelum - Ensino e Inovação'.

b) 'Caelum - Ensino e Inovação' e 'Caelum - Ensino e Inovação'.

c) 'Caelum' e 'Caelum'.

d) 'Caelum - Ensino e Inovação' e 'Caelum'.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String vazio = null;
4         String full = "Bem-vindo " + vazio;
5         System.out.println(full);
6     }
7 }
```

a) Não compila pois vazio é nulo.

b) Não compila por outro motivo.

c) Compila e imprime "Bem-vindo ".

d) Compila e imprime "Bem-vindo vazio".

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String vazio;
4         String full = "Bem-vindo " + vazio;
5         System.out.println(full);
6     }
7 }
```

a) Não compila pois vazio é nulo.

b) Não compila por outro motivo.

c) Compila e imprime "Bem-vindo ".

d) Compila e imprime "Bem-vindo vazio".

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     String vazio;
3     public static void main(String[] args) {
4         String full = "Bem-vindo " + vazio;
5         System.out.println(full);
6     }
7 }
```

a) Não compila pois vazio é nulo.

b) Não compila por outro motivo.

c) Compila e imprime “Bem-vindo “.

d) Compila e imprime “Bem-vindo vazio”.

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     static String vazio;
3     public static void main(String[] args) {
4         String full = "Bem-vindo " + vazio;
5         System.out.println(full);
6     }
7 }
```

a) Não compila pois vazio é nulo.

b) Não compila por outro motivo.

c) Compila e imprime “Bem-vindo “.

d) Compila e imprime “Bem-vindo vazio”.

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s = null
4         String s2 = new String(s);
5         System.out.println(s2);
6     }
7 }
```

a) Não compila ao tentar invocar o construtor.

b) Compila e não imprime nada.

c) Compila e imprime `null`.

d) Compila e dá erro de execução ao tentar criar a segunda `String`.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s = "estudando para a certificação";
4         System.out.println(s.substring(3, 6));
5     }
6 }
```

a) Não compila: caractere com acento e cedilha dentro de uma String.

b) Não compila: `substring` é `subString`.

c) Compila e imprime "uda".

d) Compila e imprime "tuda".

e) Compila e imprime "tud".

11) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s2 = new String(null);
4         System.out.println(s2);
5     }
6 }
```

a) Não compila ao tentar invocar o construtor.

b) Compila e não imprime nada.

c) Compila e imprime `null`.

d) Compila e dá erro de execução ao tentar criar a `String`.

12) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int valor = 10;
4         int dividePor = 4;
5         double resultado = valor / dividePor;
6         System.out.println(valor + dividePor +
7                                    " são os valores utilizados.");
8         System.out.println(resultado + " é o resultado");
9     }
10 }
```

a) Imprime os números 10, 4 e 2.5.

b) Imprime os números 14 e 2.5.

c) Nenhuma das outras alternativas.

13) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s = "estudando para a certificação";
4         s.replace("e", 'a');
5         System.out.println(s);
6     }
7 }
```

a) Não compila.

b) Compila e imprime "estudando para a certificação".

c) Compila e imprime "astudando para a cartificação".

d) Compila e imprime "studando para a crtificação".

14) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s = "guilherme";
4         s.substring(0,2) = "gua";
5         System.out.println(s);
6     }
7 }
```

a) Erro de compilação.

b) Compila e imprime "guilherme".

c) Compila e imprime "gualherme".

Capítulo 5

Usando operadores e construções de decisão

5.1 Use operadores Java

Para manipular os valores armazenados das variáveis, tanto as primitivas quanto as não primitivas, a linguagem de programação deve oferecer operadores. Um dos operadores mais importantes é o que permite guardar um valor em uma variável. Esse operador é denominado **operador de atribuição**.

No Java, o símbolo = representa o operador de atribuição. Para atribuir um valor precisamos de uma variável à qual será atribuído o valor, e do valor:

```
long idade = ; // não compila, onde está o valor?
long = 15; // não compila, onde está o nome da variável?
long idade = 15; // compila
```

```
idade = 15;
// compila desde que a variável tenha sido declarada
// anteriormente
```

Para um valor ser atribuído a uma variável, ambos devem ser compatíveis. Um valor é compatível com uma variável se ele for do mesmo tipo dela ou de um tipo menos abrangente.

```
// Iniciando uma variável com o operador de atribuição "=".
int idade = 10;

// O valor literal 10 é do tipo int e a variável é do tipo long.
// Como int é menos abrangente que long essa atribuição está
// correta.
long idade = 10;
```

Procure sempre se lembrar dos tamanhos dos primitivos quando estiver fazendo a prova. `int` é um número médio, será que ele "cabe" em uma variável do tipo `long` (número grande)? Sim, logo o código compila. Mais exemplos:

```
int a = 10;       // tipos iguais
long b = 20;      // int cabe em um long
float c = 10f;   // tipos iguais
double d = 20.0f; // float cabe em um double
double e = 30.0; // tipos iguais
float f = 40.0; // erro, double não cabe em um float.
int g = 10l; // erro, long não cabe em int
float h = 10l; // inteiros cabem em decimais
double i = 20; // inteiros cabem em decimais
long j = 20f; // decimais não cabem em inteiros
```

A exceção a essa regra ocorre quando trabalhamos com tipos inteiros menos abrangentes que `int` (`byte`, `short` e `char`). Nesses casos, o compilador permite que atribuamos um valor inteiro, desde que compatível com o tipo:

```
byte b1 = 10;
byte b2 = 200; // não compila, estoura byte
```

```
char c1 = 10;
char c2 = -3; // não compila, char não pode ser negativo
```

Atribuição e referência

Quando trabalhamos com referências, temos que lembrar do polimorfismo:

```
List<String> lista = new ArrayList<String>();
```

E no caso do Java 7, quando atribuímos com *generics* podemos usar o operador diamante:

```
// operador diamante na atribuição e inicialização:
ArrayList<String> lista = new ArrayList<>();

// operador diamante e polimorfismo junto:
List<String> lista = new ArrayList<>();
```

Lembre que as atribuições em Java são por cópia de valor, sempre. No tipo primitivo, copiamos o valor, em referências a objetos, copiamos o valor da referência (não duplicamos o objeto):

```
List<String> lista = new ArrayList<>();

// copia o valor da referência, o objeto é o mesmo
List<String> lista2 = lista;
lista2.add("Guilherme");

// verdadeiro
System.out.println(lista.size() == lista2.size());

int idade = 15;

int idade2 = idade; // copia o valor
idade2 = 20;

System.out.println(idade == idade2); // falso
```

Operadores aritméticos

Os cálculos sobre os valores das variáveis primitivas numéricas são feitos com os operadores aritméticos. A linguagem Java define operadores para as principais operações aritméticas (soma, subtração, multiplicação e divisão).

```
int dois = 2;
int dez = 10;

// Fazendo uma soma com o operador "+".
int doze = dois + dez;

// Fazendo uma subtração com o operador "-".
int oito = dez - dois;

// Fazendo uma multiplicação com o operador "*".
int vinte = dois * dez;

// Fazendo uma divisão com o operador "/".
int cinco = dez / dois;
```

Além desses, há um operador para a operação aritmética "resto da divisão". Esse operador só faz sentido para variáveis primitivas numéricas inteiras.

```
int dois = 2;
int dez = 10;

// Calculando o resto da divisão de 10 por 2.
int um = dez % dois;
```

O resultado de uma operação aritmética é um valor. A dúvida que surge é qual será o tipo dele. Para descobrir o tipo do valor resultante de uma operação aritmética, devem-se considerar os tipos das variáveis envolvidas.

A regra é a seguinte: o resultado é do tipo mais abrangente entre os das variáveis envolvidas ou, no mínimo, o `int`.

```
int idade = 15;
long anos = 5;
```

```
// ok, o maior tipo era long
long daquiCincoAnos = idade + anos;

// não compila, o maior tipo era long, devolve long
int daquiCincoAnos2 = idade + anos;
```

Mas devemos lembrar da exceção: o mínimo é um `int`:

```
byte b = 1;
short s = 2;

// devolve no mínimo int, compila
int i = b + s;

// não compila, ele devolve no mínimo int
byte b2 = i + s;

// compila forçando o casting, correndo risco de perder
// informação
byte b2 = (byte) (i + s);
```

Divisão por zero

Dividir (ou usar `mod`) um inteiro por zero lança uma `ArithmeticException`. Se o operando for um `float` ou `double`, isso gera infinito positivo ou negativo (depende do sinal do operador). As classes `Float` e `Double` possuem constantes para esses valores.

```
int i = 200;
int v = 0;

// compila, mas exception
System.out.println(i / v);

// compila e roda, infinito positivo
System.out.println(i / 0.0);
```

Ainda existe o valor `NaN` (*Not a Number*), gerado pela radiciação de um número negativo e por algumas contas com números infinitos.

```
double positivoInfinito = 100 / 0.0;
double negativoInfinito = -100 / 0.0;

// número não definido (NaN)
System.out.println(positivoInfinito + negativoInfinito);
```

Comparadores

A comparação entre os valores de duas variáveis é feita através dos operadores de comparação. O mais comum é comparar a igualdade e a desigualdade dos valores. Existem operadores para essas duas formas de comparação.

- `==` igual
- `!=` diferente

Além disso, os valores numéricos ainda podem ser comparados em relação à ordem.

- `>` maior
- `<` menor
- `>=` maior ou igual
- `<=` menor ou igual

Uma comparação pode devolver dois valores possíveis: verdadeiro ou falso. No Java, uma comparação sempre devolve um valor `boolean`.

```
System.out.println(1 == 1);        // true.
System.out.println(1 != 1);        // false.
System.out.println(2 < 1);        // false.
System.out.println(2 > 1);        // true.
System.out.println(1 >= 1);        // true.
System.out.println(2 <= 1);        // false.
```

Toda comparação envolvendo valores numéricos não considera o tipo do valor. Confira somente se eles têm o mesmo valor ou não, independente de seu tipo:

```
// true.
System.out.println(1 == 1.0);

// true.
System.out.println(1 == 1);

// true. 1.0 float é 1.0 double
System.out.println(1.0f == 1.0d);

// true. 1.0 float é 1 long
System.out.println(1.0f == 1l);
```

Os valores não primitivos (referências) e os valores *boolean* devem ser comparados somente com dois comparadores, o de igualdade (`==`) e o de desigualdade (`!=`).

```
// não compila, tipo não primitivo só aceita != e ==
System.out.println("Mario" > "Guilherme");

// não compila, boolean só aceita != e ==
System.out.println(true < false);
```

Não podemos comparar tipos incomparáveis, como um `boolean` com um valor numérico. Mas podemos comparar `chars` com numéricos.

```
// não compila, boolean é boolean
System.out.println(true == 1);

// compila, 'a' tem valor numérico também
System.out.println('a' > 1);
```

Cuidado, é muito fácil comparar atribuição com comparação e uma pegadinha aqui pode passar despercebida, como no exemplo a seguir:

```
int a = 5;
System.out.println(a = 5); // não imprime true, imprime 5
```

> **Precisão**
>
> Ao fazer comparações entre números de ponto flutuante, devemos tomar cuidado com possíveis problemas de precisão. Qualquer conta com estes números pode causar um estouro de precisão, fazendo com que ele fique ligeiramente diferente do esperado. Por exemplo, `1 == (100.0 / 100)` pode não ser verdadeiro caso a divisão tenha uma precisão não exata.

Operadores lógicos

Muitas vezes precisamos combinar os valores `boolean`s obtidos, por exemplo, com comparações ou diretamente de uma variável. Isso é feito utilizando os operadores lógicos.

Em lógica, as operações mais importantes são: `e`, `ou`, `ou exclusivo` e `negação`.

```
System.out.println(1 == 1 & 1 > 2);     // false.
System.out.println(1 == 1 | 2 > 1);     // true.
System.out.println(1 == 1 ^ 2 > 1);     // false.
System.out.println(!(1 == 1));         // false.
```

Antes de terminar a avaliação de uma expressão, eventualmente, o resultado já pode ser descoberto. Por exemplo, quando aplicamos a operação lógica `e`, ao achar o primeiro termo falso não precisamos avaliar o restante da expressão.

Quando usamos esses operadores, sempre os dois lados da expressão são avaliados mesmo nesses casos em que não precisariam.

Para melhorar isso, existem os operadores de curto circuito `&&` e `||`. Quando já for possível determinar a resposta final olhando apenas para a primeira parte da expressão, a segunda não é avaliada:

```
System.out.println(1 != 1 && 1 > 2);
// false, o segundo termo não é avaliado.
```

```
System.out.println(1 == 1 || 2 > 1);
// true, o segundo termo não é avaliado.
```

A maior dificuldade com operadores de curto circuito é se a segunda parte causa efeitos colaterais (um incremento, uma chamada de método). Avaliar ou não (independente da resposta) pode influenciar no resultado final do programa.

```
public static boolean metodo(String msg) {
    System.out.println(msg);
    return true;
}

public static void main(String[] args) {
    System.out.println(1 == 2 & metodo("oi"));
    // imprime oi, depois false
    System.out.println(1 == 2 && metodo("tchau"));
    // não imprime tchau, imprime false

    int i = 10;
    System.out.println(i == 2 & i++ == 0);
    // imprime false, soma mesmo assim
    System.out.println(i);
    // imprime 11

    int j = 10;
    System.out.println(j == 2 && j++ == 0);
    // imprime false, não soma
    System.out.println(j);
    // imprime 10
}
```

Incrementos e decrementos

Para facilitar a codificação, ainda podemos ter operadores que fazem cálculos (aritméticos) e atribuição em uma única operação. Para somar ou subtrair um valor em 1, podemos usar os operadores de incremento/decremento:

```
int i = 5;
```

```
// 5 - pós-incremento, i agora vale 6
System.out.println(i++);

// 6 - pós-decremento, i agora vale 5
System.out.println(i--);

// 5
System.out.println(i);
```

E incrementos e decrementos antecipados:

```
int i = 5;

System.out.println(++i);    // 6 - pré-incremento
System.out.println(--i);    // 5 - pré-decremento
System.out.println(i);        // 5
```

Cuidado com os incrementos e decrementos em relação a **pré** e **pós**. Quando usamos pós-incremento, essa é a última coisa a ser executada. E quando usamos o pré-incremento, é sempre a primeira.

```
int i = 10;

// 10, primeiro imprime, depois incrementa
System.out.println(i++);

// 11, valor já incrementado.
System.out.println(i);

// 12, incrementa primeiro, depois imprime
System.out.println(++i);

// 12, valor incrementado.
System.out.println(i);
```

Existem ainda operadores para realizar operações e atribuições de uma só vez:

```
int a = 10;
```

```
// para somar 2 em a
a = a + 2;

//podemos obter o mesmo resultado com:
a += 2;

//exemplos de operadores:
int i = 5;

i += 10; //soma e atribui
System.out.println(i);        // 15

i -= 10; //subtrai e atribui
System.out.println(i);        // 5

i *= 3; // multiplica e atribui
System.out.println(i);        // 15

i /= 3; // divide a atribui
System.out.println(i);        // 5

i %= 2; // divide por 2, e atribui o resto
System.out.println(i);        // 1

System.out.println(i+=3); // soma 3 e retorna o resultado: 4
```

Nesses casos, o compilador ainda dá um desconto para operações com tipos teoricamente incompatíveis. Veja:

```
byte b1 = 3; // compila, dá um desconto
b1 = b1 + 4; // não compila, conta com int devolve int

byte b2 = 3; // compila, dá um desconto
b2 += 4; // compila também, compilador gente boa!
```

Esse último caso compila inclusive se passar valores absurdamente altos: `b2+=400` é diferente de `b2 = b2 + 400`. Ele faz o *casting* e roda normalmente.

Cuidado também com o caso de atribuição com o próprio autoincremento:

```
int a = 10;
a += ++a + a + ++a;
```

Como a execução é do primeiro para o último elemento das somas, temos as reduções:

```
a += ++a + a + ++a;
a = a + ++a + a + ++a;
a = 10 +11 + a + ++a;
a = 10 + 11 + 11 + ++a;
a = 10 + 11 + 11 + 12;
a = 44; // a passa a valer 44
```

Um outro exemplo de operador pós-incremento, cujo resultado é 1 e 2:

```
int j = 0;
int i = (j++ * j + j++);
System.out.println(i);
System.out.println(j);
```

Pois:

```
i = (0 * j + j++); // j vale 1
i = (0 * 1 + j++); // j vale 1
i = (0 * 1 + 1); // j vale 2
i = 1; // j vale 2
```

Podemos fazer diversas atribuições em sequência, que serão executadas da direita para a esquerda. O resultado de uma atribuição é sempre o valor da atribuição:

```
int a = 15, b = 20, c = 30;
a = b = c; // b = 30, portanto a = 30
```

Outro exemplo mais complexo:

```
int a = 15, b = 20, c = 30;
a = (b = c + 5) + 5; // c = 30, portanto b = 35, portanto a = 40
```

Operador ternário Condicional

Há também um operador para controle de fluxo do programa, como um `if`. É chamado de **operador ternário**. Se determinada condição acontecer, ele vai por um caminho, caso contrário vai por outro.

A estrutura do operador ternário é a seguinte:

variável = teste_booleano **?** valor_se_verdadeiro **:** valor_se_falso;

```
int i = 5;
System.out.println(i == 5 ? "verdadeiro": "falso");// verdadeiro
System.out.println(i != 5 ? 1: 2);                    // 2

String mensagem = i % 2 == 0 ? "é par" : "é ímpar";
```

O operador condicional sempre tem que retornar valores que podemos usar para atribuir, imprimir etc.

Operador de referência

Para acessar os atributos ou métodos de um objeto precisamos aplicar o operador `.` (ponto) em uma referência. Você pode imaginar que esse operador navega na referência até chegar no objeto.

```
String s = new String("Caelum");

// Utilizando o operador "." para acessar um
// objeto String e invocar um método.
int length = s.length();
```

Concatenação de Strings

Quando usamos Strings, podemos usar o `+` para denotar concatenação. É a única classe que aceita algum operador fora o ponto.

Em Java, não há sobrecarga de operadores como em outras linguagens. Portanto, não podemos escrever nossas próprias classes com operadores diversos.

StringBuilder

A concatenação de Strings é um *syntax sugar* que o próprio compilador resolve. No código compilado, na verdade, é usado um `StringBuilder`.

Precedência

Não é necessário decorar a precedência de todos operadores do Java, basta saber o básico, que primeiro são executados pré-incrementos/decrementos, depois multiplicação/divisão/mod, passando para soma/subtração, depois os *shifts* (`<<`, `>>`, `>>>`) e, por último, os pós-incrementos/decrementos.

As questões da certificação não entram em mais detalhes que isto.

Pontos importantes

- Na atribuição de um valor para uma variável primitiva, o valor deve ser do mesmo tipo da variável ou de um menos abrangente.

  EXCEÇÃO À REGRA: Para os tipos `byte`, `short` e `char`, em atribuições com literais do tipo `int`, o compilador verifica se o valor a ser atribuído está no range do tipo da variável.

- Toda variável não primitiva está preparada somente para armazenar referências para objetos que sejam do mesmo tipo dela.
- Toda comparação e toda operação lógica devolve `boolean`.
- O resultado de toda operação aritmética é no mínimo `int` ou do tipo da variável mais abrangente que participou da operação.
- A comparação de valores numéricos não considera os tipos dos valores.
- As referências e os valores `boolean` só podem ser comparados com == ou != .
- Toda atribuição é por cópia de valor.

Observação: O recurso do *autoboxing* permite fazer algumas operações diferentes envolvendo variáveis não primitivas. Discutiremos sobre autoboxing adiante.

Casting de tipos primitivos

Não podemos atribuir a uma variável de um tipo um valor que não é compatível com ela:

```
double d = 3.14;
int i = d;
```

Só podemos fazer essas atribuições se os valores forem *compatíveis*. Compatível é quando um tipo cabe em outros, e ele só cabe se o *range* (alcance) dele for mais amplo que o do outro.

byte -> **short** -> **int** -> **long** -> **float** -> **double**

char -> **int**

Se estivermos convertendo de um tipo que vai da esquerda para a direita nessa tabelinha, não precisamos de casting, a **autopromoção** fará o serviço por nós.

Se estamos indo da direita para a esquerda, precisamos do *casting* e não importam os valores que estão dentro. Exemplo:

```
double d = 0;
float  f = d;
```

Esse código não compila sem um casting! O casting é a maneira que usamos para moldar uma variável de um tipo em outro. Nós estamos avisando o compilador que sabemos da possibilidade de perda de precisão ou truncamento, mas nós realmente queremos fazer isso:

```
float  f = (float) d;
```

Podemos fazer casting entre ponto flutuante e inteiro, o resultado será o número truncado, sem as casas decimais:

```
double d = 3.1415;
int i = (int) d; // 3
```

Dica

Não é preciso decorar a sequência int->long->float etc. Basta lembrar os alcances das variáveis. Por exemplo, o `char` tem dois bytes e guarda um número positivo. Será então que posso atribuir um `char` a um `short`? Não, pois um `short` tem 2 bytes, e usa meio a meio entre os números positivos e negativos.

1) Qual código a seguir compila?

a)
```
short s = 10;
char c = s;
```

b)
```
char c = 10;
long l = c;
```

c)
```
char c = 10;
short s = c;
```

2) Faça contas com diferentes operandos:

```
int i1 = 3/2;
double i2 = 3/2;
double i3 = 3/2.0;

long x = 0; double d = 0;
double zero = x + d;
System.out.println(i1 + i2 + i3 + x + d + zero);
```

Qual o resultado?

a) 3

b) 3.5

c) 4

d) 4.5

3) O código a seguir pode lançar um `NullPointerException`. Como evitar isso mantendo a mesma lógica?

```
void metodo(Carro c) {
    if(c != null & c.getPreco() > 100000) {
        System.out.println("possivel sequestro");
    }
}
```

a) Trocando `!=` por `==`

b) Trocando `>` por `<`

c) Trocando `&` por `|`

d) Trocando `&` por `&&`

4) Alguns testes interessantes com tipos primitivos:

```
int i = (byte) 5;
long l = 3.0;
float f = 0.0;
char c = 3;
char c2 = -2;
```

Quais compilam?

a) i, f e c

b) i, f, c e c2

c) i, f e c2

d) i e c

e) f e c

f) f e c2

g) i e l

h) l, f e c

i) i, c e c2

5) A expressão a seguir pode ser reduzida, como podemos fazer?

```
if ((trem && !carro) || (!trem && carro)) {
    // ....
}
```

- Trocando para usar um operador `&` e um `|`
- Trocando para usar dois operadores `&` e um `|`
- Trocando para usar um operador `!` e um `^`
- Trocando para usar um operador `^`
- Removendo os parênteses
- Removendo o `||` do meio
- Removendo os `!`

6) Imprima a divisão por 0 de números inteiros e de números com ponto flutuante:

```
System.out.println(3 / 0);
System.out.println(3 / 0.0);
System.out.println(3.0 / 0);
System.out.println(-3.0 / 0);
```

Quais os resultados?

7)
```
1 class Xyz {
2   public static void main(String[] args) {
3       int y;
4       for(int x = 0; x<10; ++x) {
5         y = x % 5 + 2;
6       }
7       System.out.println(y);
8   }
9 }
```

Qual o resultado desse código?

a) Erro de compilação na linha 3

b) Erro de compilação na linha 7

c) 1

d) 2

e) 3

f) 4

g) 5

h) 6

8)
```
1 class Teste {
2     public static void main(String[] args){
3         byte b = 1;
4         int i = 1;
5         long l = 1;
6         float f = 1.0;
7     }
8 }
```

O código:

a) Não compila a linha 3 pois "1" é `int` e não pode ser colocado em um `byte`

b) Não compila a linha 4 pois "1" é `long` e não pode ser colocado em um `int`

c) Não compila a linha 5 pois "1" é `int` e não pode ser colocado em um `long`

d) Não compila a linha 6 pois "1.0" é `double` e não pode ser colocado em um `float`

e) Todas as linhas compilam

9)
```
class $_o0o_$ {
    public static void main(String[] args) {
        int $$ = 5;
        int __ = $$++;
        if (__ < ++$$ || __-- > $$)
```

```
                System.out.print("A");

            System.out.print($$);
            System.out.print(__);
        }
    }
```

O estranho código:

a) Não compila por causa do nome da classe

b) Não compila por causa dos nomes das variáveis

c) Compila mas dá erro na execução

d) Compila, roda e imprime A76

e) Compila, roda e imprime A75

f) Compila, roda e imprime A74

g) Compila, roda e imprime 76

1) O que acontece com o seguinte código? Compila? Roda?

```
public class Teste{
    public static void main(String[] args) {
        byte b1 = 5;
        byte b2 = 3;
        byte b3 = b1 + b2;
    }
}
```

1) O que acontece com seguinte código?

```
1 public class Teste{
2     public static void main(String[] args) {
3         byte b1 = 127;
4         byte b2 = -128;
5         byte b3 = b1 + b2;
```

```
6         System.out.println(b3);
7     }
8 }
```

a) Não compila por um erro na linha 3

b) Não compila por um erro na linha 4

c) Não compila por um erro na linha 5

d) Compila e imprime -1

1)
```
1 public class Teste {
2     public static void main(String[] args) {
3         int i;
4         for (i = 0; i < 5; i++) {
5             if (++i % 3 == 0) {
6                 break;
7             }
8         }
9         System.out.println(i);
10     }
11 }
```

Qual é o resultado do código:

a) imprime 1

b) imprime 2

c) imprime 3

d) imprime 4

2) Considerando o mesmo código da questão anterior, e se trocarmos para pós-incremento dentro do `if`?

```
1 public class Teste {
2     public static void main(String[] args) {
3         int i;
4         for (i = 0; i < 5; i++) {
```

```
5             if (i++ % 3 == 0) {
6                 break;
7             }
8         }
9         System.out.println(i);
10    }
11 }
```

Qual é o resultado?:

a) imprime 1

b) imprime 2

c) imprime 3

d) imprime 4

3) Qual é o resultado do seguinte código:

```
public class Teste {
    public static void main(String[] args) {
        int i;
        for (i = 0; i < 5; i++) {
            if (++i % 3 == 0) {
                break;
            }
        }
        System.out.println(i);
    }
}
```

4) E se trocarmos o pré-incremento para pós-incremento (`i++`)?

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         byte b1 = 100;
4         byte b2 = 131;
```

```
5         System.out.println(b1);
6     }
7 }
```

a) Compila e imprime um número positivo.

b) Compila e imprime um número negativo.

c) Compila e dá uma exception de estouro de número.

d) Compila e imprime um número que não sabemos dizer ao certo.

e) Compila e imprime “Not A Number”.

f) Não compila.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         char c = 65;
4         char c2 = -3;
5         System.out.println(c + c2);
6     }
7 }
```

a) Não compila nas duas declarações de `char`.

b) Não compila nas três linhas dentro do método `main`.

c) Não compila somente na declaração de `c2`.

d) Não compila somente na soma de caracteres.

e) Compila e roda, imprimindo 62.

f) Compila e roda, imprimindo um outro valor.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         char c = 65;
4         char c2 = 68 - 65;
```

```
5         System.out.println(c + c2);
6     }
7 }
```

a) Não compila nas duas declarações de `char`.

b) Não compila nas três linhas dentro do método `main`.

c) Não compila somente na declaração de `c2`.

d) Não compila somente na soma de caracteres.

e) Compila e roda, imprimindo 62.

f) Compila e roda, imprimindo um outro valor.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         double resultado = 15 / 0;
4         System.out.println(resultado);
5     }
6 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String resultado = "divisao dá: " + 15 / 0.0;
4         System.out.println(resultado);
5     }
6 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         System.out.println(1==true);
4     }
5 }
```

a) Não compila.

b) Compila e imprime verdadeiro.

c) Compila e imprime falso.

5.2 Use parentêses para sobrescrever a precedência de operadores

Às vezes desejamos alterar a ordem de precedência de uma linha, e nesses instantes usamos os parênteses:

```
int a = 15 * 4 + 1; // 15 * 4 = 60, depois 60 + 1 = 61
int b = 15 * (4 + 1); // 4 + 1 = 5, depois 15 * 5 = 75
```

Devemos tomar muito cuidado na concatenação de `String` e precedência:

```
System.out.println(15 + 0 + " é cento e cinquenta");
// 15 é cento e cinquenta
System.out.println(15 + (0 + " é cento e cinquenta"));
// 150 é cento e cinquenta

System.out.println(("guilherme" + " silveira").length());
// 18
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String resultado = ("divisao dá: " + 15) / 0.0;
        System.out.println(resultado);
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(((!(true==false))==true ? 1 : 0)==0);
    }
}
```

a) Imprime `true`.

b) Imprime `false`.

c) Não compila.

d) Imprime 1.

e) Imprime 0.

5.3 Teste a igualdade entre Strings e outros objetos usando == e equals()

Observe o seguinte código que cria duas Strings:

```
String nome1 = new String("Mario");
String nome2 = new String("Mario");
```

Como já estudamos anteriormente, o operador == é utilizado para comparação. Neste caso, como se tratam de objetos, irá comparar as duas referências e ver se apontam para o mesmo objeto:

```
String nome1 = new String("Mario");
String nome2 = new String("Mario");

System.out.println(nome1 == nome2); // imprime false
```

Até aqui tudo bem. Mas vamos alterar um pouco nosso código, mudando a maneira de criar nossas Strings, e rodar novamente:

```
String nome1 = "Mario";
String nome2 = "Mario";

System.out.println(nome1 == nome2); // o que imprime?
```

Ao executar o código, vemos que ele imprime `true`. O que aconteceu?

Pool de Strings

O Java mantém um *pool* de objetos do tipo `String`. Antes de criar uma nova String, primeiro o Java verifica neste pool se uma String com o mesmo conteúdo já existe; caso sim, ele a reutiliza, evitando criar dois objetos exatamente iguais na memória. Como as duas referências estão apontando para o mesmo objeto do pool, o == retorna `true`.

Mas por que isso não aconteceu antes, com nosso primeiro exemplo? O Java só coloca no pool as Strings criadas usando **literais**. Strings criadas com o operador `new` não são colocadas no pool automaticamente.

```
String nome1 = "Mario"; //será colocada no pool
String nome2 = new String("Mario");
/*
"Mario" é colocado, mas nome2 é outra
referência, não colocada no pool
*/
```

Sabendo disso, temos que ter cuidado redobrado quando comparando Strings usando o operador ==:

```
String s1 = "string";
String s2 = "string";
String s3 = new String("string");

System.out.println(s1 == s2); // true, mesma referencia
System.out.println(s1 == s3); // false, referências diferentes
System.out.println(s1.equals(s3)); // true, mesmo conteúdo
```

Repare que, mesmo sendo instâncias diferentes, quando comparadas usando o método `equals`, o retorno é `true`, caso o conteúdo das Strings seja o mesmo.

Quando concatenamos literais, a String resultante também será colocada no pool.

```
String ab = "a" + "b";
System.out.println("ab" == ab); // true
```

Mas isso é verdade **apenas usando literais em ambos os lados da concatenação**. Se algum dos objetos não for um literal, o resultado será um novo objeto, que não estará no pool:

```
String a = "a";
String ab = a + "b"; //usando uma referência e um literal
System.out.println("ab" == ab); // false
```

Sabemos que Strings são imutáveis, e que cada método chamado em uma String retorna uma nova String, sem alterar o conteúdo do objeto original. Esses objetos resultantes de retornos de métodos não são buscados no pool, são novos objetos:

```
String str = "um texto qualquer";
String txt1 = "texto";
String txt2 = x.substring(3, 8); //cria uma nova string
System.out.println(txt1 == txt2); // false
System.out.println(txt.equals(x.substring(3, 8)); // true
```

OS MÉTODOS DE STRING SEMPRE CRIAM NOVOS OBJETOS?

Nem sempre. Se o retorno do método for exatamente o conteúdo atual do objeto, nenhum objeto novo é criado:

```
String str = "HELLO WORLD";
String upper = str.toUpperCase();          // já está maiúscula
String subs = str.substring(0,11);         // string completa
System.out.println(str == upper);          // true
System.out.println(str == subs);           // true
System.out.println(str == str.toString()); // true
```

Contando Strings

Uma questão recorrente na prova é contar quantos objetos do tipo `String` são criados em um certo trecho de código. Veja o código a seguir e tente descobrir quantos objetos String são criados:

```
String h = new String ("hello ");
String h1 = "hello ";
String w = "world";

System.out.println("hello ");
System.out.println(h1 + "world");
System.out.println("Hello " == h1);
```

E então? Vamos ver passo a passo:

```
//Cria 2 objetos, um literal (que vai para o pool) e o outro
//com o new
String h = new String ("hello ");

//nenhum objeto criado, usa o mesmo do pool
String h1 = "hello ";
//novo objeto criado e inserido no pool
String w = "world";

```

```
10 //nenhum  objeto criado, usa do pool
11 System.out.println("hello ");
12
13 //criado um novo objeto resultante da concatenação,
14 // mas este não vai para o pool
15 System.out.println(h1 + "world");
16
17 //Novo objeto criado e colocado no pool (Hello com H maiúsculo).
18 System.out.println("Hello " == h1);     // 1
```

Logo temos 5 Strings criadas.

Cuidado com String já colocadas no pool

Para descobrir se uma String foi criada e colocada no pool é necessário prestar muita atenção ao contexto do código e ao enunciado da questão. A String só é colocada no pool na primeira execução do trecho de código. Cuidado com questões que criam Strings dentro de métodos, ou que dizem em seu enunciado que o método já foi executado pelo menos uma vez:

```
1  public class Testes {
2      public static void main(String[] args) {
3          for(int i = 0; i< 10; i++)
4              System.out.println(metodo());
5      }
6
7      private static String metodo() {
8          String x = "x";
9          return x.toString();
10     }
11 }
```

Ao executar essa classe, apenas **um** objeto String será criado. O único lugar onde a String é criada é na linha 8 do código.

O método equals

Para comparar duas referências, podemos sempre usar o operador ==. Dada a classe `Cliente`:

```
class Cliente {
    private String nome;
    Cliente(String nome) {
        this.nome = nome;
    }
}

Cliente c1 = new Cliente("guilherme");
Cliente c2 = new Cliente("mario");
System.out.println(c1==c2); // false
System.out.println(c1==c1); // true

Cliente c3 = new Cliente("guilherme");
System.out.println(c1==c3);
// false, pois não é a mesma
// referência: são objetos diferentes na memória
```

Para comparar os objetos de uma outra maneira, que não através da referência, podemos utilizar o método `equals`, cujo comportamento padrão é fazer a simples comparação com o ==:

```
Cliente c1 = new Cliente("guilherme");
Cliente c2 = new Cliente("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Cliente c3 = new Cliente("guilherme");
System.out.println(c1.equals(c3));
// false, pois não é a mesma
// referência: são objetos diferentes na memória
```

Isso é, existe um método em `Object` que você pode reescrever para definir um **critério de comparação de igualdade**. Classes como `String`, `Integer` e muitas outras possuem esse método reescrito, assim `new Integer(10) == new Integer(10)` dá `false`, mas `new Integer(10).equals(Integer(10))` dá `true`.

É interessante reescrever esse método quando você julgar necessário um critério de igualdade diferente que o `==` retorna. Imagine o caso de nosso `Cliente`:

```
class Cliente {
    private String nome;
    Cliente(String nome) {
        this.nome = nome;
    }

    public boolean equals(Object o) {
        if (! (o instanceof Cliente)) {
            return false;
        }
        Cliente outro = (Cliente) o;
        return this.nome.equals(outro.nome);
    }
}
```

O método `equals` não consegue tirar proveito do `generics`, então precisamos receber `Object` e ainda verificar se o tipo do objeto passado como argumento é realmente uma `Cliente` (o contrato do método diz que você deve retornar `false`, e não deixar lançar exception em um caso desses). Agora sim, podemos usar o método `equals` como esperamos:

```
Cliente c1 = new Cliente("guilherme");
Cliente c2 = new Cliente("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Cliente c3 = new Cliente("guilherme");
System.out.println(c1.equals(c3)); // true
```

Cuidado ao sobrescrever o método `equals`: ele deve ser público, e deve receber `Object`. Caso você receba uma referência a um objeto do tipo `Cliente`, seu método não está sobrescrevendo aquele método padrão da classe `Object`, mas sim crianod um novo método (overload). Por polimorfismo o compilador fará funcionar neste caso pois o compilador fará a

conexão ao método mais específico, entre `Object` e `Cliente`, ele escolherá o método que recebe `Cliente`:

```
class Cliente {
    private String nome;
    Cliente(String nome) {
        this.nome = nome;
    }

    public boolean equals(Cliente outro) {
        return this.nome.equals(outro.nome);
    }
}

Cliente c1 = new Cliente("guilherme");
Cliente c2 = new Cliente("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Cliente c3 = new Cliente("guilherme");
System.out.println(c1.equals(c3)); // true
System.out.println(c1.equals((Object) c3));
// false, o compilador não sabe que Object é cliente,
// invoca o equals tradicional, e azar do desenvolvedor
```

Mas caso você use alguma biblioteca (como a API de coleções e de `ArrayList` do Java), o resultado não será o esperado.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String s1 = "s1";
4         String s2 = "s" + "1";
5         System.out.println(s1==s2);
6         System.out.println(s1==("" + s2));
7     }
8 }
```

a) Não compila.

b) Compila e imprime `true`, `false`.

c) Compila e imprime `true`, `true`.

d) Compila e imprime `false`, `false`.

e) Compila e imprime `false`, `true`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s1 = "s1";
        String s2 = s1.substring(0, 1) + s1.substring(1,1);
        System.out.println(s1==s2);
        System.out.println(s1.equals(s2));
    }
}
```

a) Não compila.

b) Compila e imprime `true`, `false`.

c) Compila e imprime `true`, `true`.

d) Compila e imprime `false`, `false`.

e) Compila e imprime `false`, `true`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
            String s1 = "s1";
            String s2 = s1.substring(0, 2);
            System.out.println(s1==s2);
            System.out.println(s1.equals(s2));
    }
}
```

a) Não compila.

b) Compila e imprime `true`, `false`.

c) Compila e imprime `true, true`.

d) Compila e imprime `false, false`.

e) Compila e imprime `false, true`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B extends C{}
class C {
    int x;
    public boolean equals(C c) {
        return c.x==x;
    }
}
class A {
    public static void main(String[] args) {
        C a = new C();
        C b = new B();
        a.x = 1;
        b.x = 1;
        System.out.println(a==b);
        System.out.println(a.equals(b));
    }
}
```

a) Não compila.

b) Compila e imprime `true, false`.

c) Compila e imprime `true, true`.

d) Compila e imprime `false, false`.

e) Compila e imprime `false, true`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B extends C{}
class D {
    int x;
}
```

```
5  class C {
6      int x;
7      public boolean equals(Object c) {
8          return c.x==x;
9      }
10 }
11 class A {
12     public static void main(String[] args) {
13         C a = new C();
14         C b = new D();
15         a.x = 1;
16         b.x = 1;
17         System.out.println(a==b);
18         System.out.println(a.equals(b));
19     }
20 }
```

a) Não compila.

b) Compila e imprime `true`, `false`.

c) Compila e imprime `true`, `true`.

d) Compila e imprime `false`, `false`.

e) Compila e imprime `false`, `true`.

5.4 Utilize o if e if/else

Imagine um programa que aceita comandos do usuário, ou seja, um sistema interativo. De acordo com os dados que o usuário passar, o programa se comporta de maneiras diferentes e, consequentemente, pode dar respostas diferentes.

O programador, ao escrever esse programa, deve ter recursos para definir o comportamento para cada possível comando do usuário, em outras palavras, para cada situação. Com isso, o programa será capaz de tomar decisões durante a execução com o intuito de mudar o fluxo de execução.

As linguagens de programação devem oferecer aos programadores maneiras para *controlar o fluxo de execução dos programas*. Dessa forma, os

programas podem tomar decisões que afetam a sequência de comandos que serão executados.

if / else

A maneira mais simples de controlar o fluxo de execução é definir que um determinado trecho de código deve ser executado quando uma condição for verdadeira.

Por exemplo, suponha um sistema de login. Ele deve verificar a autenticidade do usuário para permitir ou não o acesso. Isso pode ser implementado com um **if/else** do Java.

```
boolean autentico = true;
if (autentico) {
    System.out.println("Usuario aceito");
} else {
    System.out.println("Usuario incorreto");
}
```

A sintaxe do `if` é a seguinte:

```
if (CONDICAO) {
    // CODIGO 1
} else {
    // CODIGO 2
}
```

A condição de um `if` **sempre** tem que ser um valor booleano:

```
1 if(1 - 2) { } // erro, numero inteiro
2
3 if(1 < 2) {} //ok, resulta em true
4
5 boolean valor = true;
6 if (valor == false) {} // ok, mas resulta em false
7
8 if (valor) {} // ok, valor é boolean
```

Atenção dobrada ao código a seguir:

```
int a = 0, b = 1;

if(a = b) {
    System.out.println("iguais");
}
```

Esta é uma pegadinha bem comum. Repare que não estamos fazendo uma **comparação** aqui, e sim, uma **atribuição** (um único `=`). O resultado de uma atribuição é sempre o valor atribuído, no caso, um inteiro. Logo, este código não compila, pois passamos um inteiro para a condição do `if`.

A única situação em que um código assim poderia funcionar é caso a variável atribuída seja do tipo `boolean`, pois o resultado da atribuição será `boolean`:

```
boolean a = true;

if(a = false) {
    System.out.println("Falso!");
}
```

Neste caso, o código compila, mas não imprime nada. Após a atribuição, o valor da variável `a` é `false`, e o `if` não é executado.

Caso só tenhamos um comando dentro do `if` ou `else`, as chaves são opcionais:

```
if(!resultado)
    System.out.println("Falso!");
else
    System.out.println("Verdadeiro!");
```

Caso não tenhamos nada para ser executado em caso de condição `false`, não precisamos declarar o `else`:

```
boolean autentico = true;
if (autentico)
    System.out.println("Usuario aceito");
```

Mas sempre temos que ter algum código dentro do `if`, se não o código não compila:

```
boolean autentico = true;
if (autentico)
else // erro
    System.out.println("Acesso negado");
```

Na linguagem Java, **não** existe o comando **elseif**. Para conseguir o efeito do "elseif", os `ifs` são colocados dentro dos `else`.

```
if (CONDICAO1) {
    // CODIGO 1
} else if (CONDICAO2) {
    // CODIGO 2
} else {
    // CODIGO 3
}
```

Grande parte das perguntas sobre estruturas de `if/else` são pegadinhas, usando a indentação como forma de distração:

```
boolean autentico = true;
if (autentico)
    System.out.println("Usuario aceito");
else
    System.out.println("Usuario incorreto");
    System.out.println("Tente novamente");
```

A mensagem `"Tente novamente"` sempre é impressa, independente do valor da variável `autentico`.

Esse foi um exemplo bem simples, vamos tentar algo mais complicado. Tente determinar o que é impresso:

```
int valor = 100;
if (valor > 200)
if (valor <400)
if (valor > 300)
    System.out.println("a");
else
    System.out.println("b");
else
    System.out.println("c");
```

E então? "c"? Vamos reindentar o código para ver se fica mais fácil:

```
int valor = 100;
if (valor > 200)
    if (valor <400)
        if (valor > 300)
            System.out.println("a");
        else
            System.out.println("b");
    else
        System.out.println("c");
```

É sempre complicado analisar código não indentado ou mal indentado, e esse recurso é usado extensivamente em várias questões durante a prova, fique esperto!

Unreachable Code e Missing return

Um código Java não compila se o compilador perceber que aquele código não será executado sob hipótese alguma:

```
class Teste {
    public int metodo() {
        return 5;
        System.out.println("Quando isso será executado?");
    }
}
```

```
Teste.java:10: unreachable statement
        System.out.println("Quando isso será executado?");
                  ^
```

O código após o `return` não será nunca executado. Esse código não compila. Vamos ver alguns outros exemplos:

```
class Teste {
    public int metodo(int x) {
        if(x > 200) {
            return 5;
        }
```

```
    }
}
```

Este também não compila. O que será retornado se `x` for `<= 200`?

```
Teste.java:12: missing return statement
    }
        ^
1 error
```

Vamos modificar o código para que ele compile:

```
class Teste {
    public int metodo(int x) {
        if(x > 200) {
            return 5;
        }
        throw new RuntimeException();
    }
}
```

Apesar de não estarmos retornando nada caso o `if` seja falso, o Java percebe que nesse caso uma exceção será disparada. A regra é: todos os caminhos possíveis devem retornar o tipo indicado pelo método, ou lançar exceção.

Em um `if`, essa expressão compila normalmente:

```
if(false) {.... } //compila, apesar de ser unreachable code
```

São pequenos detalhes, tome cuidado para não cair nessas pegadinhas.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         if(args.length > 0)
4             System.out.println("Um ou mais argumentos");
5         else
6             System.out.println("0");
7     }
8 }
```

a) Não compila: `length` é método.

b) Não compila: faltou chaves no `if` e `else`.

c) Se invocarmos sem argumentos, imprime `0`.

d) Nunca imprimirá `0`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B{
2      final boolean valor = false;
3  }
4  class A {
5      public static void main(String[] args) {
6          B b = new B();
7          if(b.valor = true) {
8              System.out.println("verdadeiro");
9          }
10     }
11 }
```

a) Não compila.

b) Compila e imprime `verdadeiro`.

c) Compila e não imprime nada..

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int quantidade = 15;
4          if(quantidade=15) {
5              System.out.println("sim");
6          } else {
7              System.out.println("nao");
8          }
9      }
10 }
```

a) Não compila.

b) Imprime `sim`.

c) Imprime `não`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          if(args.length==1)
4              System.out.println("Um");
5          elseif(args.length==2)
6              System.out.println("Dois");
7          elseif(args.length==3)
8              System.out.println("Três");
9          else
10             System.out.println("Quatro");
11     }
12 }
```

a) Não compila.

b) Roda e imprime "Um" quando passamos um argumento.

c) Roda e imprime "Três" quando passamos 4 argumentos.

d) Roda e não imprime nada quando passamos nenhum argumento.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          String nome = args[0];
4          if(nome.equals("guilherme"))
5              System.out.println(nome);
6              System.out.println("bom");
7          else
8              System.out.println("melhor ainda");
9              System.out.println(nome);
10     }
11 }
```

a) Erro de compilação no `if`.

b) Erro de compilação no `else`.

c) Compila e imprime o nome e "bom" caso o primeiro argumento seja `guilherme`.

d) Compila e dá erro de execução caso não passe nenhum argumento na linha de comando.

5.5 Utilize o switch

Suponha que um programa tenha que reagir diferentemente para três casos possíveis. Por exemplo, suponha que o usuário possa passar três valores possíveis: `1`, `2` e `3`. Se for `1`, o programa deve imprimir "PRIMEIRA OPCAO", se for `2`, "SEGUNDA OPCAO", e se for `3`, "TERCEIRA OPCAO".

Isso pode ser implementado com `if/else`. Mas, há uma outra possibilidade. O Java, assim como outras linguagens de programação, oferece o comando **switch**. Ele permite testar vários casos de uma maneira diferente do `if/else`.

```
int opcao = 1;
switch (opcao) {
    case 1:
        System.out.println("PRIMEIRA OPCAO");
    case 2:
        System.out.println("SEGUNDA OPCAO");
    case 3:
        System.out.println("TERCEIRA OPCAO");
}
```

O `switch` tem uma sintaxe cheia de detalhes e uma semântica pouco intuitiva. Vamos analisar cada um desses detalhes separadamente para ficar mais simples.

O argumento do `switch` dever ser uma variável compatível com o tipo primitivo `int`, um *wrapper* de um tipo menor que `Integer`, uma **String** ou um **enum**. Enums Não são cobrados nessa prova, então vamos focar apenas nos outros dois casos.

O valor de cada `case` deve ser compatível com o tipo do argumento do `switch`, caso contrário será gerado um erro de compilação na linha do `case` inválido.

```
//argumento do switch int, e cases int
int valor = 20;
switch (valor){
    case 10 : System.out.println(10);
    case 20 : System.out.println(20);
}

//Argumento String, e cases String
String s = "Oi";
switch (s) {
    case "Oi": System.out.println("Olá");
    case "Hi": System.out.println("Hello");
}

//Argumento Byte, e cases byte
Byte b = 10;
switch (b) {
    case 10: System.out.println("DEZ");
}

//argumento do switch int, e cases string, não compila
int mix = 20;
switch (mix){
    case "10" : System.out.println(10);
    case "20" : System.out.println(20);
}
```

Cuidado pois `switch` de `double` não faz sentido conforme a lista de argumentos que citamos compatíveis com o `switch`!

```
double mix = 20;
switch (mix){  // não compila
    case 10.0 : System.out.println(10);
    case 20.0 : System.out.println(20);
}
```

Você pode usar qualquer tipo primitivo menor que um `int` como argumento do `switch`, desde que os tipos dos cases sejam compatíveis:

```
//argumento do switch byte
byte valor = 20;

switch (valor){
    // Apesar de ser inteiro, 10 cabe em um byte, o compilador
    // fará o cast automaticamente
    case 10 :
        System.out.println(10);
}

switch (valor){
    // Neste caso, o número é muito grande, o compilador não
    // fará o cast e teremos um erro de compilação pois os tipos
    // são incompatíveis
    case 32768 :  //erro
        System.out.println(10);
}
```

Em cada `case`, só podemos usar como valor um literal, uma variável `final` atribuída com valor literal, ou expressões envolvendo os dois. Nem mesmo `null` é permitido:

```
int valor = 20;
final int CINCO = 5;
int trinta = 30;

switch (valor) {
    case CINCO: // constante
        System.out.println(5);
    case 10: // literal
        System.out.println(10);
    case CINCO * 4: // operação com constante e literal
        System.out.println(20);
    case trinta: // erro, variável
        System.out.println(30);
    case trinta + CINCO: //erro, operação envolvendo variável
```

```
        System.out.println(35);
    case null: // erro, null em case
        System.out.println("null");
}
```

Constantes em cases

Para ser considerada uma constante em um `case`, a variável, além de ser final, também deve ter sido inicializada durante a sua declaração. Inicializar a variável em outra linha faz com que ela não possa ser usada como valor em um `case`:

```
int v = 10;
final int DEZ = 10;
final int VINTE; // final, mas não inicializada
VINTE = 20; // inicializada

switch (v) {
case DEZ:
    System.out.println("DEZ!");
    break;
case VINTE: //erro
    System.out.println("DEZ!");
    break;
}
```

O `switch` também aceita a definição de um caso padrão, usando a palavra `default`. O caso padrão é aquele que deve ser executado se nenhum `case` “bater”.

```
int opcao = 4;
switch (opcao) {
    case 1:
        System.out.println("PRIMEIRA OPCAO");
    case 2:
        System.out.println("SEGUNDA OPCAO");
```

```
    case 3:
        System.out.println("TERCEIRA OPCAO");
    default:
        System.out.println("CASO PADRAO");
}
```

Um detalhe sobre a sintaxe do `default` é que ele pode aparecer antes de um ou de diversos `case`s. Desta forma:

```
int opcao = 4;
switch(opcao) {
    case 1:
        System.out.println("PRIMEIRA OPCAO");
    case 2:
        System.out.println("SEGUNDA OPCAO");
    default:
        System.out.println("CASO PADRAO");
    case 3:
        System.out.println("TERCEIRA OPCAO");
}
```

Um comportamento contraintuitivo do `switch` é que, quando executado, se algum `case` "bater", tudo que vem abaixo é executado também, todos os `case`s e o `default`, se ele estiver abaixo. Esse comportamento também vale se cair no `default`. Por exemplo, o código anterior imprime:

```
CASO PADRAO
TERCEIRA OPCAO
```

Com esse comportamento, podemos inclusive criar `case`s sem nenhum bloco de código dentro:

```
int v = 1;
switch(v){
    case 1:
    case 2:
    case 3:
        System.out.println("Até 3");
}
```

Para mudar esse comportamento e não executar o que vem abaixo de um `case` que bater ou do `default`, é necessário usar o comando `break` em cada `case`.

```
int opcao = 4;
switch(opcao) {
    case 1:
        System.out.println("PRIMEIRA OPCAO");
        break;
    case 2:
        System.out.println("SEGUNDA OPCAO");
        break;
    default:
        System.out.println("CASO PADRAO");
        break;
    case 3:
        System.out.println("TERCEIRA OPCAO");
        break;
}
```

Neste caso, só será impresso "TERCEIRA OPCAO".

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int tamanho = args.length;
4         switch(tamanho) {
5             case 1:
6                 System.out.println("1");
7             case 2:
8                 System.out.println("2");
9             default:
10                System.out.println("mais argumentos");
11        }
12    }
13 }
```

a) Não compila.

b) Ao rodar sem argumentos joga uma exception..

c) Ao rodar com dois argumentos, imprime somente “2”..

d) Ao rodar com 5 argumentos, imprime “mais argumentos”.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int tamanhoEsperado = 1;
4         int tamanho = args.length;
5         switch(tamanho) {
6             case tamanhoEsperado:
7                 System.out.println("1");
8                 break;
9             default:
10                System.out.println("cade o argumento?");
11        }
12    }
13 }
```

a) Não compila.

b) Ao rodar sem argumentos joga uma exception.

c) Ao rodar com um argumento, imprime somente “1”.

d) Ao rodar com 5 argumentos, imprime “cade o argumento?”

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         switch("Guilherme") {
4             case "Guilherme":
5                 System.out.println("Guilherme");
6                 break;
7             case "42":
8                 System.out.println("42");
9             default:
```

```
10                  System.out.println("Outro nome");
11          }
12      }
13  }
```

a) Não compila, pois um número não pode ser comparado com `String`.

b) Compila e imprime `Guilherme`.

c) Não compila, pois o código do `case 42` e `default` nunca serão executados.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int count = args.length;
4          switch(count) {
5              case 0 {
6                  System.out.println("nenhum");
7                  break;
8              } case 1 {
9              } case 2 {
10                 System.out.println("ok");
11             } default {
12                 System.out.println("default");
13             }
14         }
15     }
16 }
```

a) Erro de compilação.

b) Se rodar com 1 argumento, imprime `ok` e mais uma mensagem.

c) Se rodar com 1 argumento, não imprime nada.

d) Se rodar com 5 argumentos, imprime `default`.

e)

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         switch(10) {
4             case < 10:
5                 System.out.println("menor");
6             default:
7                 System.out.println("igual");
8             case > 10:
9                 System.out.println("maior");
10        }
11    }
12 }
```

a) Erro de compilação.

b) Compila e imprime "igual".

c) Compila e imprime "igual" e "maior".

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         switch(10) {
4             case 10:
5                 System.out.println("a");
6                 break;
7                 System.out.println("b");
8             default:
9                 System.out.println("c");
10            case 11:
11                System.out.println("d");
12        }
13    }
14 }
```

a) Não compila.

b) Imprime `a` e `b` e `c` e `d`.

c) Imprime `a`.

Capítulo 6

Criando e usando arrays

6.1 Declare, instancie, inicialize e use um array uni-dimensional

As linguagens de programação, normalmente, fornecem algum recurso para o armazenamento de variáveis em memória sequencial. No Java, os arrays permitem esse tipo de armazenamento.

Um array é um objeto que armazena sequencialmente "uma porção" de variáveis de um determinado tipo. É importante reforçar que os arrays são objetos. Uma referência para um objeto array deve ser armazenada em uma variável do tipo array.

A prova de certificação verifica se o candidato está apto a manipular tanto arrays de tipos primitivos quanto de tipos não primitivos.

Os quatro pontos importantes sobre arrays são:

- Declarar
- Inicializar
- Acessar
- Percorrer

Arrays de tipos primitivos

Declaração:

Para declarar um array, é utilizado `[]` logo após ao tipo das variáveis que desejamos armazenar ou logo após ao nome da variável.

```
// Declaração de um array para guardar variáveis do tipo int.
int[] idades;

// Declaração de um array para guardar variáveis do tipo double.
double pesos[];

// Declaração de um array para guardar variáveis do tipo long.
long []pesos;

// Declaração de um array para guardar variáveis do tipo long.
long[]tamanhos;

// Perceba as formas de declarar um array.
```

Inicialização:

Como um array é um objeto, a inicialização envolve a criação de um objeto. O **new**, operador que cria objetos, é utilizado para construir um array. Se você não executa o **new**, qual o valor padrão? Para atributos, é **null**, e para variáveis locais, não há valor, como qualquer outra variável de referência:

```
public class Clientes {

    int[] idades;

    public static void main(String[] args) {
```

```
        Clientes c = new Clientes();
        System.out.println(c.idades); // imprime null
    }
}
public class Produtos {

    public static void main(String[] args) {
        int[] precos;
        System.out.println(precos); // nao compila, não foi
                                    // inicializada
    }
}
```

E como instancio um array?

```
// Inicialização do array idades.
idades = new int[10];

// Inicialização do array pesos.
pesos = new double[50];
```

Na inicialização, é definida a capacidade do array, ou seja, a quantidade de variáveis que ele terá. Quando falarmos em tamanho de um array, estaremos nos referindo à sua capacidade.

Cada variável guardada em um array é iniciada implicitamente no momento em que o array é criado. Os valores atribuídos às variáveis são os valores `default`.

```
// Imprime 0 pois esse é o valor default para int.
System.out.println(idades[0]);
```

E temos alguns casos extremos:

```
//compila e roda
int[] numeros = new int[0];

//compila, mas joga NegativeArraySizeException
numeros = new int[-1];
```

Durante a declaração de uma referência para um array, temos a oportunidade de criá-lo de uma maneira mais fácil se já sabemos o que queremos colocar dentro:

```
int numeros;
numeros = new int[]{1, 2, 3, 4, 5};

Carro[] carros = new Carro[]{new Carro(), null, new Carro()};
```

Não passamos o tamanho e fazemos a declaração dos elementos entre chaves e separados por vírgula. Os arrays terão tamanho 5 e 3, respectivamente.

E se a declaração e a inicialização estiverem na **mesma linha** podemos simplificar ainda mais:

```
int[] numeros = {1, 2, 3, 4, 5};
```

Mas temos que tomar um pouco de cuidado com esse modo mais simples de declarar o array. Só podemos fazer como no exemplo anterior quando **declaramos e inicializamos** o array na mesma linha. Se fizermos a declaração e a inicialização em linhas separadas, o código não compila:

```
int [] numeros = {1, 2, 3, 4, 5}; //compila
int [] numeros2;
numeros2 = {1, 2, 3, 4, 5}; // não compila
```

Se desejamos inicializar posteriormente, devemos adicionar o operador `new` para poder iniciar o array em outra linha:

```
int [] numeros2;
numeros2 = new int [] {1, 2, 3, 4, 5}; // compila
```

Acesso

As posições de um array são indexadas (numeradas) de `0` até a capacidade do array menos um. Para acessar uma das variáveis do array, é necessário informar sua posição.

```
// Coloca o valor 10 na primeira variável do array idades.
int idades[] = new int[10];
idades[0] = 10;

// Coloca o valor 73.14 na última variável do array pesos.
double pesos[] = new double[50];
pesos[49] = 73.14;
```

O que acontece se alguém tentar acessar uma posição que não existe?

```
// Erro de execução ao tentar acessar um posição que não existe.
// ArrayIndexOutOfBoundsException
pesos[50] = 88.4;
```

Será gerado um erro de execução (não de compilação). A *exception* lançada pelo Java é `ArrayIndexOutOfBoundsException`.

Percorrendo

Supondo que a capacidade de um array qualquer seja `100`, os índices desse array variam de `0` até `99`, ou seja, de `0` até a capacidade menos um.

O tamanho de um array é definido na inicialização e fica guardado no próprio array, podendo ser recuperado posteriormente.

Para recuperar o tamanho ou a capacidade de um array, é utilizado um atributo chamado `length` presente em todos os arrays.

```
for (int i = 0; i < idades.length; i++) {
    idades[i] = i;
}
```

No `for` tradicional, as posições de um array são acessadas através dos índices. Dessa forma, é possível, inclusive, modificar os valores que estão armazenados no array.

Porém, em determinadas situações, é necessário apenas ler os valores de um array sem precisar modificá-los. Nesse caso, pode ser utilizado o `for` introduzido na versão 5 do Java.

```
for(int idade : idades){
    System.out.println(idade);
}
```

Não há índices no `for` do Java 5. Ele simplesmente percorre os valores. Assim ele não permite modificar o array facilmente.

Array de referências

Em cada posição de um array de tipos não primitivos é guardada uma variável não primitiva. Esse é um fato fundamental.

```
// Declarando e iniciando um array de Prova
Prova[] provas = new Prova[10];
```

Lembrando que o `new` inicia as variáveis implicitamente e que o valor padrão para variáveis não primitivas é `null`, todas as dez posições do array desse código estão `null` imediatamente após o `new`.

```
// Erro de execução ao tentar aplicar o operador "."
// em  uma referência com valor null.
// NullPointerException
provas[0].tempo = 10;
```

Para percorrer um array de tipos não primitivos, podemos utilizar um laço:

```
for (int i = 0; i < provas.length; i++){
    provas[i] = new Prova();
    provas[i].tempo = 210;
}

for (Prova prova : provas){
    System.out.println(prova.tempo);
}
```

Caso a classe `Prova` seja abstrata, devido ao polimorfismo é possível adicionar filhas de `Prova` nesse array: o polimorfismo funciona normalmente, portanto funciona igualmente para interfaces.

```
class Prova {
}
class ProvaPratica extends Prova {
}
```

```
class Test {
    public static void main(String[] args) {
        Prova[] provas = new Prova[2];
        provas[0] = new Prova();
        provas[1] = new ProvaPratica();
    }
}
```

Uma vez que o array de objetos é sempre baseado em referências, lembre-se que um objeto não será copiado, mas somente sua referência passada:

```
Cliente guilherme = new Cliente();
guilherme.setNome("guilherme");

Cliente[] clientes = new Clientes[10];
clientes[0] = guilherme;

System.out.println(guilherme.getNome()); // guilherme
System.out.println(clientes[0].getNome()); // guilherme

guilherme.setNome("Silveira");

System.out.println(guilherme.getNome()); // silveira
System.out.println(clientes[0].getNome()); // silveira
```

Casting de arrays

Não há *casting* de arrays de tipo primitivo, portanto não adianta tentar:

```
int[] valores = new int[10];
long[] vals = valores; // não compila
```

Já no caso de referências, por causa do polimorfismo é possível fazer a atribuição sem casting de um array para outro tipo de array:

```
String[] valores = new String[2];
valores[0] = "Certificação";
valores[1] = "Java";

Object[] vals = valores;
```

```
for(Object valor : vals) {
    System.out.println(valor); // Certificação e depois Java
}
```

E o casting compila normalmente mas, ao executarmos, um array de `Object` não é um array de String e levamos uma `ClassCastException`:

```
Object[] valores = new Object[2];
valores[0] = "Certificação";
valores[1] = "Java";
String[] vals = (String[]) valores;
for(Object valor : vals) {
    System.out.println(valor);
}
```

Isso pois a classe dos dois é distinta e a classe pai de `array de string` não é um array de objeto, e sim, um `Object` (lembre-se: todo array herda de `Object`):

```
Object[ ] objetos = new Object[ 2 ];
String[ ] strings = new String[ 2 ];
System.out.println(objetos.getClass().getName());
// [ Ljava.lang.Object;
System.out.println(strings.getClass().getName());
// [ Ljava.lang.String;

System.out.println(strings.getClass().getSuperclass());
// java.lang.Object
```

1) Escolha a opção que não compila:

a) `int[] x;`

b) `int x[];`

c) `int[]x;`

d) `int [] x;`

e) `int[] x;`

f) `[]int x;`

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int x[] = new int[30];
4         int y[] = new int[3] {0,3,5};
5     }
6 }
```

   a) A primeira linha não compila.
   b) A segunda linha não compila.
   c) O código compila e roda.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, em relação as linhas dentro do método main:

```
1 class A {
2     public static void main(String[] args) {
3         int x[] = new int[0];
4         int x[] = new int[] {0,3,5};
5         int x[] = {0,3,5};
6     }
7 }
```

   a) A primeira e segunda linhas não compilam.
   b) A segunda e terceira linhas não compilam.
   c) Somente a terceira linha não compila.
   d) O programa compila e roda, dando uma exception.
   e) O programa compila e roda, imprimindo nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int x[] = new int[3];
4         for(int i=x.length;i>=0;i--) x[i]=i*2;
```

```
5           System.out.println("Fim!");
6       }
7 }
```

a) O programa não compila

b) O programa imprime `Fim`.

c) O programa compila e dá erro em execução.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int x[] = new int[3];
4         for(x[1]=x.length-1;x[0]==0;x[1]--) {
5             x[x[1]]=-5;
6             System.out.println(x[1]);
7         }
8     }
9 }
```

a) Não compila.

b) Compila, imprime alguns números e dá uma `Exception`.

c) Compila e não imprime nada.

d) Compila e imprime `2`.

e) Compila e imprime `-5`.

f) Compila e imprime `2, -5`.

g) Compila e imprime `2, -5, -5`.

h) Compila e imprime `2, 1, -5`.

i) Compila e imprime `-5, -5`.

j) Dá exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int x[] = new int[3];
4         for(x[1]=x.length-1;x[1]>=0;x[1]--) {
5             x[x[1]]=-5;
6             System.out.println(x[1]);
7         }
8     }
9 }
```

a) Não compila.

b) Compila, imprime alguns números e dá uma `Exception`.

c) Compila e não imprime nada.

d) Compila e imprime `2`.

e) Compila e imprime `-5`.

f) Compila e imprime `2, -5`.

g) Compila e imprime `2, -5, -5`.

h) Compila e imprime `2, 1, -5`.

i) Compila e imprime `-5, -5`.

j) Dá exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String[] valores = new String[2];
4         valores[0] = "Certificação";
5         valores[1] = "Java";
6         Object[] vals = (Object[]) valores;
7         vals[1] = "Daniela";
8         System.out.println(vals[1].equals(valores[1]));
9     }
10 }
```

a) O código não compila.

b) O código compila e dá erro em execução.
c) O código compila e imprime `false`.
d) O código compila e imprime `true`.

8) Quais das maneiras adiante são declarações e inicializações válidas para um array?

a) `int[] array = new int[10];`
b) `int array[] = new int[10];`
c) `int[] array = new int[];`
d) `int array[] = new int[];`
e) `int[] array = new int[2]{1, 2};`
f) `int[] array = new int[]{1, 2};`
g) `int[] array = int[10];`
h) `int[] array = new int[1, 2, 3];`
i) `int array[] = new int[1, 2, 3];`
j) `int array[] = {1, 2, 3};`

6.2 Declare, instancie, inicialize e use um array multi-dimensional

Podemos generalizar a ideia de array para construir arrays de duas dimensões, em outras palavras, **array de arrays**. Analogamente, podemos definir arrays de quantas dimensões quisermos.

Declaração:

```
// Um array de duas dimensões.
int[][] tabela;

// Um array de três dimensões.
int[][] cubo[];
```

```
// Um array de quatro dimensões.
int[] [][]hipercubo[];

// Perceba que as dimensões podem ser definidas do lado
// esquerdo ou direito da variável.
```

Inicialização:

```
// Inicializando a primeira dimensão com 10 e a segunda com 15
tabela = new int[10][15];

// Inicializando a primeira dimensão com 10 e deixando as outras
// para serem iniciadas depois
cubo = new int[10][][];

// Inicializando com valores
int[][] teste = new int[][]{{1,2,3},{3,2,1},{1,1,1}};
```

Acesso:

```
// Acessando a posição (0,1)
System.out.println(tabela[0][1]);
```

Podemos criar um array que não precisa ser "quadrado", ele pode ter tamanhos estranhos:

```
int[][] estranha = new int[2][];
estranha[0] = new int[20];
estranha[1] = new int[10];
for(int i=0;i<estranha.length;i++) {
    System.out.println(estranha[i].length); // imprime 20 e 10
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int zyx[][]=new int[3];
4         int[]x=new int[20];
```

```
5         int[]y=new int[10];
6         int[]z=new int[30];
7         zyx[0]=x;
8         zyx[1]=y;
9         zyx[2]=z;
10        System.out.println(zyx[2].length);
11    }
12 }
```

a) Não compila, erro ao declarar `zyx`.

b) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

c) Compila e dá erro ao tentar imprimir o tamanho do array.

d) Compila e imprime 10.

e) Compila e imprime 20.

f) Compila e imprime 30.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int zyx[][]=new int[3][];
4          int[]x=new int[20];
5          int[]y=new int[10];
6          int[]z=new int[30];
7          zyx[0]=x;
8          zyx[1]=y;
9          zyx[2]=z;
10         System.out.println(zyx[2].length);
11     }
12 }
```

a) Não compila, erro ao declarar `zyx`.

b) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

c) Compila e dá erro ao tentar imprimir o tamanho do array.

d) Compila e imprime 10.

e) Compila e imprime 20.

f) Compila e imprime 30.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int zyx[][]=new int[3][10];
4          int[]x=new int[20];
5          int[]y=new int[10];
6          int[]z=new int[30];
7          zyx[0]=x;
8          zyx[1]=y;
9          zyx[2]=z;
10         System.out.println(zyx[2].length);
11     }
12 }
```

a) Não compila, erro ao declarar `zyx`.

b) Não compila, erro ao atribuir arrays de tamanho diferente de `10` em `zyx`.

c) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

d) Compila e dá erro ao tentar imprimir o tamanho do array.

e) Compila e imprime 10.

f) Compila e imprime 20.

g) Compila e imprime 30.

4)
```
class Teste {
    public static void main(String[] args){
        int[] idades = new int[10];
        idades[0] = 1.0;

        int[10][10] tabela = new int[10][10];

        int[][][] cubo = new int[][][];
    }
}
```

a) O código não compila.

b) O código compila e dá erro em execução.

c) O código compila e roda.

Compila? Roda?

6.3 Declare e use uma ArrayList

Nesta prova, veremos somente a `ArrayList`, uma lista que usa internamente um array. Rápida no método `get`, pois sua estrutura interna permite acesso aleatório (*random access*) em tempo constante.

Jamais se esqueça de importar a `ArrayList`:

```
import java.util.ArrayList;
```

O primeiro passo é criar uma `ArrayList` vazia de `Strings`:

```
ArrayList<String> nomes = new ArrayList<String>();
```

A `ArrayList` herda diversos métodos abstratos e concretos e veremos vários deles aqui, dentre esses, os principais para a certificação, vindos da interface `Collection`.

Por exemplo, para adicionar itens, fazemos:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");
```

Para remover e verificar a existência do mesmo na lista:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");

System.out.println(nomes.contains("java")); // true
System.out.println(nomes.contains("c#")); // false

// true, encontrado e removido
```

```
boolean removido = nomes.remove("java");

System.out.println(nomes.contains("java")); // false
System.out.println(nomes.contains("c#")); // false
```

Note que o `remove` remove somente a primeira ocorrência daquele objeto.

Podemos também verificar o tamanho de nossa `ArrayList`:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");
System.out.println(nomes.size()); // imprime 2
```

E convertê-la para um array:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");

Object[] nomesComoString = nomes.toArray();
```

Caso desejarmos um array de String, devemos indicar isso ao método `toArray` de duas formas diferentes:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");

String[] nomes2 = nomes.toArray(new String[0]);
String[] nomes3 = nomes.toArray(new String[nomes.size()]);
```

Ambas passam um array de String: o primeiro menor e o segundo com o tamanho suficiente para os elementos. Se ele possui o tamanho suficiente, ele mesmo será usado, enquanto que, se o tamanho não é suficiente, o `toArray` cria um novo array do mesmo tipo.

Além disso, podemos adicionar uma coleção inteira em outra:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
nomes.add("java");

ArrayList<String> paises = new ArrayList<String>();
paises.add("coreia");
paises.add("brasil");

ArrayList<String> tudo = new ArrayList<String>();
tudo.addAll(nomes);
tudo.addAll(paises);
System.out.println(tudo.size()); // imprime 4
```

Outros métodos são específicos da interface `List` e recebem uma posição específica onde você quer colocar ou remover algo do array usado na `ArrayList`. O método `get` devolve o elemento na posição desejada, lembrando que começamos sempre com o:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
System.out.println(nomes.get(0)); // imprime certificação
```

Já o método `add` foi sobrecarregado para receber a posição de inclusão:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("certificação");
System.out.println(nomes.get(0)); // imprime certificação

nomes.add(0, "java");
System.out.println(nomes.get(0)); // imprime java
System.out.println(nomes.get(1)); // imprime certificação
```

O mesmo acontece para o método `remove`:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("java");
nomes.add("certificação");

String removido = nomes.remove(0); // retorna java
System.out.println(nomes.get(0)); // imprime certificação
```

E o método `set`, que serve para alterar o elemento em determinada posição:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("java");
nomes.set(0, "certificação");

System.out.println(nomes.get(0)); // imprime certificação
System.out.println(nomes.size()); // imprime 1
```

Os métodos `indexOf` e `lastIndexOf` retornam a primeira ou a última posição que possui o elemento desejado. Caso esse elemento não esteja na lista, ele retorna -1:

```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("guilherme");
nomes.add("mario");
nomes.add("paulo");
nomes.add("mauricio");
nomes.add("adriano");
nomes.add("alberto");
nomes.add("mario");

System.out.println(nomes.indexOf("guilherme")); // 0
System.out.println(nomes.indexOf("mario")); // 1
System.out.println(nomes.indexOf("joao")); // -1
System.out.println(nomes.lastIndexOf("mario")); // 6
System.out.println(nomes.lastIndexOf("joao")); // -1
```

Iterator e o enhanced for

A interface `Iterator` define uma maneira de percorrer coleções. Isso é necessário porque, em coleções diferentes de `List`, não possuímos métodos para pegar o enésimo elemento. Como, então, percorrer todos os elementos de uma coleção?

- `hasNext`: retorna um booleano indicando se ainda há elementos a serem percorridos por esse iterador;

- `next`: pula para o próximo elemento, devolvendo-o;
- `remove`: remove o elemento atual da coleção.

O código que costuma aparecer para percorrer uma coleção é o seguinte:

```
Collection<String> strings = new ArrayList<String>();
Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {
    String atual = iterator.next();
    System.out.println(atual);
}
```

O `enhanced-for` também pode ser usado nesse caso:

```
Collection<String> strings = new ArrayList<String>();
for (String atual : strings) {
    System.out.println(atual);
}
```

O método equals em coleções

A maioria absoluta das coleções usa o método `equals` na hora de buscar por elementos, como nos métodos `contains` e `remove`. Se você deseja ser capaz de remover ou buscar elementos, terá que provavelmente sobrescrever o método `equals` para refletir o conceito de igualdade em que está interessado, e não somente a igualdade de referência (implementação padrão do método).

Cuidado ao tentar sobrescrever o método `equals`, se você escrevê-lo recebendo um tipo específico em vez de `Object`, não o estará sobrescrevendo, e o `ArrayList` continuará invocando o código antigo, a implementação padrão de `equals`!

ArrayList e referências

Vale lembrar que Java sempre trabalha com referências para objetos, e não cria cópias de objetos cada vez que os atribuímos a uma variável ou referência:

```
Cliente guilherme = new Cliente();
guilherme.setNome("guilherme");
```

```
ArrayList<Cliente> clientes = new ArrayList<Cliente>();
clientes.add(guilherme);

System.out.println(guilherme.getNome()); // guilherme
System.out.println(clientes.get(0).getNome()); // guilherme

guilherme.setNome("Silveira");

System.out.println(guilherme.getNome()); // Silveira
System.out.println(clientes.get(0).getNome()); // Silveira
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         ArrayList<String> c = new ArrayList<String>();
4         c.add("a");
5         c.add("c");
6         System.out.println(c.remove("a"));
7     }
8 }
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e ao rodar imprime `a`.

d) Compila e ao rodar imprime `true`.

e) Compila e ao rodar imprime `false`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.util.ArrayList;
2 class A {
3     public static void main(String[] args) {
4         ArrayList<String> c = new ArrayList<String>();
5         c.add("a");
```

```
6         c.add("c");
7         System.out.println(c.remove("a"));
8     }
9 }
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e ao rodar imprime `a`.

d) Compila e ao rodar imprime `true`.

e) Compila e ao rodar imprime `false`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.util.ArrayList;
2 class A {
3     public static void main(String[] args) {
4         ArrayList<String> c = new ArrayList<String>();
5         c.add("a");
6         c.add("a");
7         System.out.println(c.remove("a"));
8         System.out.println(c.size());
9     }
10 }
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e ao rodar imprime `a` e 0.

d) Compila e ao rodar imprime `true` e 0.

e) Compila e ao rodar imprime `a` e 1.

f) Compila e ao rodar imprime `true` e 1.

g) Compila e ao rodar imprime `a` e 2.

h) Compila e ao rodar imprime `true` e 2.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("a");list.add("b");
        list.add("a");list.add("c");
        list.add("a");list.add("b");
        list.add("a");
        System.out.println(list.lastIndexOf("b"));
    }
}
```

a) Não compila.

b) Compila e imprime -1.

c) Compila e imprime 0.

d) Compila e imprime 1.

e) Compila e imprime 2.

f) Compila e imprime 3.

g) Compila e imprime 4.

h) Compila e imprime 5.

i) Compila e imprime 6.

j) Compila e imprime 7.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> l = new ArrayList<String>();
        l.add("a");
        l.add("b");
        l.add(1,"amor");
        l.add(3,"baixinho");
        System.out.println(l);
        String[] array = l.toArray();
```

```
11          System.out.println(array[2]);
12      }
13  }
```

a) Não compila.

b) Compila e imprime "amor".

c) Compila e imprime "b".

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  import java.util.ArrayList;
2  class A {
3      public static void main(String[] args) {
4          ArrayList<String> a = new ArrayList<String>();
5          ArrayList<String> b = new ArrayList<String>();
6          ArrayList<String> c = new ArrayList<String>();
7          b.add("a");c.add("c");
8          b.add("b");c.add("d");
9          a.addAll(b);
10         a.addAll(c);
11         System.out.println(a.get(0));
12         System.out.println(a.get(3));
13     }
14 }
```

a) Não compila

b) Compila e imprime `a` e `d`.

c) Compila e imprime `c` e `b`.

d) Compila e não sabemos a ordem em que os elementos serão impressos.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  import java.util.ArrayList;
2  class A {
3      public static void main(String[] args) {
4          ArrayList<String> a = new ArrayList<String>();
```

```
5         a.add("a", 0);
6         a.add("b", 0);
7         a.add("c", 0);
8         a.add("d", 0);
9         System.out.println(a.get(0));
10        System.out.println(a.get(1));
11        System.out.println(a.get(2));
12        System.out.println(a.get(3));
13    }
14 }
```

a) Não compila.

b) Compila e imprime `abcd`.

c) Compila e imprime `dcba`.

d) Compila e imprime `adcb`.

e) Compila e imprime `bcda`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.util.*;
2 class A {
3   public static void main(String[] args) {
4     ArrayList<String> a = new ArrayList<String>();
5     a.add(0,"b");
6     a.add(0,"a");
7     for(Iterator<String> i=a.iterator();i.hasNext();i.next()) {
8         String element = i.next();
9         System.out.println(element);
10    }
11 }
12 }
```

a) Não compila.

b) Compila e imprime `a`.

c) Compila e imprime `a` e `b`.

d) Compila e imprime `b` e `a`.

e) Compila e imprime `b`.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> ss = new ArrayList<String>();
        ss.add("a");
        ss.add("b");
        ss.add("c");
        ss.add("d");

        for(String s:ss){
            if(s.equals("c")) s = "b";
            else if(s.equals("b")) s= "c";
        }
        for(String s:ss) System.out.println(s);
    }
}
```

a) Não compila, `s` é `final` por padrão.

b) Compila e imprime `a, c, b, d`.

c) Compila e imprime `a, b, c, d`.

d) Compila e imprime `a, c, c, d`.

e) Compila e imprime `a, c, b, d`.

Capítulo 7

Usando laços

7.1 Crie e use laços do tipo while

Outra maneira de controlar o fluxo de execução de um programa é definir que um determinado trecho de código deve executar várias vezes, como uma repetição ou um laço.

Uma linguagem como o Java oferece alguns tipos de laços para o programador escolher. O comando `while` é um deles.

```
int i = 1;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

A sintaxe do `while` é a seguinte:

```
while (CONDICAO) {
    // CODIGO
}
```

Assim como no `if`, a condição de um bloco `while` deve ser um booleano. Da mesma maneira, se o bloco de código tiver apenas uma linha, podemos omitir as chaves:

```
int i = 0;
while( i < 10)
    System.out.println(i++);
```

O corpo do `while` é executado repetidamente até que a condição se torne falsa. Em outras palavras, enquanto a condição for verdadeira.

É necessário tomar cuidado para não escrever um `while` infinito, ou seja, um laço que não terminaria se fosse executado.

```
int i = 1;
//Quando fica false?
while(i < 10){
    System.out.println(i);
}
```

Em casos em que é explícito que o loop será infinito, o compilador é esperto e não deixa compilar caso tenha algum código após o laço:

```
class A {
    int a() {
        while(true) { //nunca fica false
            System.out.println("Faz algo");
        }
        return 1; // não compila, nunca chegará aqui
    }
}
```

Mesmo que a condição use uma variável, pode ocorrer um erro de compilação, caso a variável seja final:

```
class A {
    int a() {
```

```
        final boolean RODANDO = true;
        while(RODANDO) {
            System.out.println("Faz algo");
        }
        return 1; // não compila, nunca chegará aqui
    }
}
```

Agora, caso a variável não seja final, o compilador não tem como saber se o valor irá mudar ou não, por mais explícito que possa parecer, e o código compila normalmente:

```
class A {
    int a() {
        boolean rodando = true; // não final
        while(rodando) {
            System.out.println("Faz algo");
        }
        return 1;
        // compila, não tem como saber se o valor de rodando
        // vai mudar
    }
}
```

Caso um laço nunca seja executado, também teremos um erro de compilação:

```
//unreachable statement, não compila.
while(false) { //código aqui }

//unreachable statement, não compila.
while(1 > 2) { //código aqui }
```

Lembre-se que o compilador só consegue analisar operações com literais ou com constantes. No caso a seguir, o código compila, mesmo nunca sendo executado:

```
int a = 1;
int b = 2;
```

```
while(a > b){ //compila, mas nunca executa
    System.out.println("OI");
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int a = 10;
4         while(a>100) a++;
5         System.out.println(a);
6     }
7 }
```

a) Não compila pois nunca entra no loop.

b) Compila e imprime 99.

c) Compila e imprime 100.

d) Compila e imprime 101.

e) Compila e imprime outro valor.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         boolean rodar = true;
4         while(rodar) {
5             System.out.println(rodar);
6         }
7         System.out.println("Terminou");
8     }
9 }
```

a) Transformar a variável em `final` faz o código compilar.

b) Colocar uma linha dentro do laço que faz `rodar = false` faz o código compilar.

c) O código compila e roda em loop infinito.

d) O código compila e roda, após algumas passagens pelo laço ele imprime uma exception e para.

7.2 Crie e use laços do tipo for, incluindo o enhanced for

Observando um pouco os códigos que utilizam `while`, dá para perceber que eles são formados por quatro partes: inicialização, condição, comandos e atualização.

```
int i = 1; // Inicialização
while (i < 10) { // Condição
    System.out.println(i); // Comandos
    i++; // Atualização
}
```

A inicialização é importante para que o laço execute adequadamente. Mesmo com essa importância, a inicialização fica separada do `while`.

A atualização é fundamental para que não aconteça um "loop infinito". Porém, a sintaxe do `while` não a coloca em evidência.

Há um outro laço que coloca em destaque a inicialização, a condição e a atualização. Esse laço é o `for`.

```
for (int i = 1; i < 10; i++) {
    System.out.println(i);
}
```

O `for` tem três argumentos separados por `;`. O primeiro é a inicialização, o segundo, a condição, e o terceiro, a atualização.

A inicialização é executada somente uma vez no começo do `for`. A condição é verificada no começo de cada rodada (iteração). A atualização é executada no fim de cada iteração.

Todos os três argumentos do `for` são opcionais. Desta forma, você poderia escrever o seguinte código:

```
for(;;){
    // CODIGO
}
```

O que acontece com esse laço? Para responder essa pergunta é necessário saber quais são os "valores default" colocados nos argumentos do `for`, quando não é colocado nada pelo programador. A inicialização e a atualização ficam realmente vazias. Agora, a condição recebe por padrão o valor `true`. Então, o código anterior depois de compilado fica assim:

```
//loop infinito
for (;true;){
    // CODIGO
}
```

Nos exemplos anteriores, basicamente o que fizemos na inicialização foi declarar e inicializar apenas uma variável qualquer. Porém, é permitido declarar diversas variáveis de um mesmo tipo ou inicializar diversas variáveis.

Na inicialização, não é permitido declarar variáveis de tipos diferentes. Mas é possível inicializar variáveis de tipos diferentes. Veja os exemplos:

```
// Declarando três variáveis do tipo int e inicializando as três.
// Repare que o "," separa as declarações e inicializações.
for (int i = 1, j = 2, k = 3;;){
    // CODIGO
}

// Declarando três variáveis de tipos diferentes
int a;
double b;
boolean c;

// Inicializando as três variáveis já declaradas
for (a = 1, b = 2.0, c = true;;){
    // CODIGO
}
```

Na atualização, é possível fazer diversas atribuições separadas por `,`.

```
//a cada volta do laço, incrementamos o i e decrementamos o j
for (int i=1,j=2;; i++,j--){
    //código
}
```

Como já citamos anteriormente, não é possível inicializar variáveis de tipos diferentes:

```
for (int i=1, long j=0; i< 10; i++){ // erro
    //código
}
```

No campo de condição, podemos passar qualquer expressão que resulte em um `boolean`. São exatamente as mesmas regras do `if` e `while`.

No campo de atualização, não podemos só usar os operadores de incremento, podemos executar qualquer trecho de código:

```
for (int i = 0; i < 10; i += 3) { //somatório
//código
}

for (int i = 0; i < 10; System.out.println(i++)) { // bizarro
//código
}
```

Enhanced for

Quando vamos percorrer uma coleção de objetos ou um array, podemos usar uma versão simplificada do `for` para percorrer essa coleção de maneira simplificada. Essa forma simplificada é chamada de *enhanced for*, ou *foreach* :

```
int[] numeros = {1,2,3,4,5,6};
for (int num : numeros) { //enhanced for
    System.out.println(num);
}
```

A sintaxe é mais simples, temos agora 2 partes dentro da declaração do `for`:

```
for(VARIAVEL : COLEÇÃO){
    CODIGO
}
```

Nesse caso, declaramos uma variável que irá receber cada um dos membros da coleção ou array que estamos percorrendo. O próprio `for` irá a cada iteração do laço atribuir o próximo elemento da lista à variável. Seria o equivalente a fazer o seguinte:

```
int[] numeros = {1,2,3,4,5,6};

for( int i=0; i < numeros.length; i++){
    int num = numeros[i]; //declaração da variável e atribuição
    System.out.println(num);
}
```

Se fosse uma *collection*, o código fica mais simples ainda se comparado com o `for` original:

```
ArrayList<String> nomes = //lista com vários nomes

//percorrendo a lista com o for simples
for(Iterator<String> iterator = nomes.iterator();
    iterator.hasNext();){
    String nome = iterator.next();
    System.out.println(nome);
}

//percorrendo com o enhanced for
for (String nome : nomes) {
    System.out.println(nome);
}
```

Existem, porém, algumas limitações no *enhanced for*. Não podemos, por exemplo, modificar o conteúdo da coleção que estamos percorrendo usando a variável que declaramos:

```
ArrayList<String> nomes = //lista com vários nomes

//tentando remover nomes da lista
for (String nome : nomes) {
    nome = null;
}
```

```
//o que imprime abaixo?
for (String nome : nomes) {
    System.out.println(nome);
}
```

Ao executar esse código, você perceberá que a coleção não foi modificada, nenhum elemento mudou de valor para `null`.

Outra limitação é que não há uma maneira natural de saber em qual iteração estamos, já que não existe nenhum contador. Para saber em qual linha estamos, precisaríamos de um contador externo. Também não é possível percorrer duas coleções ao mesmo tempo, já que não há um contador centralizado. Para todos esses casos, é recomendado usar o `for` simples.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         for(;;) {
4             System.out.println("a");
5         }
6         System.out.println("b");
7     }
8 }
```

a) Não compila.

b) Compila e imprime `a` infinitamente.

c) Compila e imprime `b`.

d) Compila e imprime `a`, depois `b`, depois para.

e) Compila, imprime `a` diversas vezes e depois dá um `StackOverflowError`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
```

```
3         for(;false;) {
4             System.out.println("a");
5             break;
6         }
7         System.out.println("b");
8     }
9 }
```

a) Não compila.

b) Compila e imprime b.

c) Compila, imprime a e b.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2   public static void main(String[] args) {
3     for(int i=0, int j=1; i<10; i++, j++) System.out.println(i);
4 }
5 }
```

a) Não compila.

b) Compila e imprime 0 até 9.

c) Compila e imprime 0 até 10.

d) Compila e imprime 1 até 10.

e) Compila e imprime 1 até 11.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         for(int i=0, j=1; i<10 ;i++, j++) System.out.println(i);
4     }
5 }
```

a) Não compila.

b) Compila e imprime 0 até 9.

c) Compila e imprime 0 até 10.

d) Compila e imprime 1 até 10.

e) Compila e imprime 1 até 11.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         for(int i=0; i<10, false; i++) {
4             System.out.println('a');
5         }
6         System.out.println('b');
7     }
8 }
```

a) Não compila.

b) Compila e imprime 'a' e 'b'.

c) Compila e imprime 'b'.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         for(int i=0; i<2; i++, System.out.println(i)) {
4             System.out.println(i);
5         }
6     }
7 }
```

a) Não compila.

b) Compila e imprime `0 1 2`.

c) Compila e imprime `0 0 1 1 2 2`.

d) Compila e imprime `0 1 1 2 2`.

e) Compila e imprime `0 1 1 2`.

7.3 Crie e uso laços do tipo do/while

Uma outra opção de laço `while` seria o `do .. while`, que é bem parecido com o `while`. A grande diferença é que a condição é testada após o corpo do *loop* ser executado pelo menos uma vez:

```
int i = 1;
do { //executa ao menos 1 vez
    System.out.println(i);
    i++;
} while (i < 10); // se der true, volta e executa novamente.
```

A condição do `do .. while` só é verificada no final de cada iteração e não no começo, como no `while`. Repare que ao final do bloco `do .. while` existe um ponto e vírgula. Esse é um detalhe que passa desapercebido muitas vezes, mas que resulta em erro de compilação se omitido:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i < 10) // não compila, faltou o ;
```

Assim como no `while`, caso tenhamos apenas uma linha, as chaves podem ser omitidas. Caso exista mais de uma linha dentro do `do .. while` e não existam chaves, teremos um erro de compilação:

```
int i = 0;
//compila normal
do
    System.out.println(i++);
while(i<10);

//erro, mais de uma linha dentro do do .. while
do
    System.out.print("o valor é: "); //erro
    System.out.println(i++);
while(i<10);
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean i = false;
        do {
            System.out.println(i);
        } while(i);
    }
}
```

a) Não compila.

b) Compila e imprime `false`.

c) Compila e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        if(args.length < 10) {
            do {
                if(args.length>2) return;
            } while(true);
        }
        System.out.println("Finalizou");
    }
}
```

a) Não compila.

b) Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Não imprime nada caso 3 a 9 argumentos. Imprime 'Finalizou' caso 10 ou mais argumentos.

c) Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Imprime 'Finalizou' caso contrário.

d) Compila e sempre entra em loop infinito.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int i = 0;
4         do System.out.println(i); while(i++>10);
5     }
6 }
```

a) Não compila.

b) Compila e não imprime nada.

c) Compila e imprime `0`.

d) Compila e imprime de `0` até `9`.

e) Compila e imprime de `0` até `10`.

f) Compila e não imprime nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int i = 0;
4         do System.out.println(i) while(i++<10);
5     }
6 }
```

a) Não compila.

b) Compila e imprime de `0` até `9`.

c) Compila e imprime de `0` até `10`.

d) Compila e não imprime nada.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int i = 0;
```

```
4           do; while(i++<10);
5       }
6   }
```

a) Não compila.

b) Compila e entra em loop infinito.

c) Compila e sai.

7.4 Compare os tipos de laços

Embora o `for`, `while` e `do .. while` sejam todos estruturas que permitam executar *loops*, existem similaridades e diferenças entre essas construções que serão cobradas na prova.

Comparando while e do .. while

No caso do `while` e `do while`, ambos são muito similares, sendo a principal diferença o fato do `do .. while` ter a condição testada somente após executar o código de dentro do *loop* pelo menos uma vez.

```
int i = 20;

//imprime 20, já que só faz o teste após a execução do código
do {
    System.out.println(i);
    i++;
} while(i < 10);

int j = 20;

//não imprime nada, já que testa antes de executar o bloco
while(j < 10){
    System.out.println(i);
    i++;
}
```

Comparando for e enhanced for

Apesar de ser mais complexo, o `for` simples é mais poderoso que o `enhanced for`. Com o `enhanced for`, não podemos:

- Percorrer mais de uma coleção ao mesmo tempo;
- Remover os elementos da coleção;
- Inicializar um array.

Caso desejemos fazer uma iteração de **leitura, por todos os elementos da coleção**, aí sim o `enhanced for` é a melhor opção.

Comparando while e for

Ambas estruturas de laço permitem executar as mesmas operações em nosso código, e são bem similares. Mas, apesar disso, existem situações em que o código ficará mais simples caso optemos por uma delas.

Geralmente optamos por usar `for` quando sabemos a quantidade de vezes que queremos que o laço seja executado. Pode ser percorrer todos os elementos de uma coleção, (onde sabemos a quantidade de vezes que o loop será executado por saber o tamanho da coleção) ou simplesmente executar o laço uma quantidade fixa de vezes.

Usamos o `while` ou `do .. while` quando não sabemos a quantidade de vezes em que o laço será executado, mas sabemos uma condição que, enquanto for verdadeira, fará com que o laço seja repetido.

O exemplo a seguir mostra um código no qual conhecemos a condição de parada, mas não faz sentido nenhum ter uma variável inicializada ou uma condição de incremento, então escolhemos um `while`:

```
while(conta.getSaldo() > 0) {
    conta.saca(1000);
}
```

Note que, caso queira contar quantas vezes foi sacado, faria sentido usar um `for`:

```
int saques;
for(saques = 0; conta.getSaldo() > 0; saques++) {
    conta.saca(1000);
}
System.out.println("Saquei " + saques + " vezes");
```

1) Qual o laço mais simples de ser usado quando desejamos iterar por duas coleções ao mesmo tempo?

   a) for
   b) while
   c) enhanced for
   d) do... while

2) Qual o melhor laço a ser usado para, dependendo do valor de um elemento, removê-lo de nossa lista?

   a) for
   b) enhanced for

3) Para todos os números entre `i` e `100` devo imprimir algo, sendo que, mesmo que `i` seja maior que `100`, devo imprimir algo pelo menos uma vez. Qual laço devo usar?

   a) enhanced for
   b) do... while
   c) while

4) Qual o laço a ser usado caso queira executar um código eternamente?

   a) enhanced for
   b) for
   c) for ou while

d) for ou while ou do...while

e) while ou do...while

5) Qual laço deve ser usado para inicializar os valores de um array?

a) enhanced for

b) for

7.5 Use break e continue

Em qualquer estrutura de laço podemos aplicar os controladores **break** e **continue**. O `break` serve para parar o laço totalmente. Já o `continue` interrompe apenas a iteração atual. Vamos ver alguns exemplos:

```
int i = 1;
while (i < 10) {
    i++;
    if (i == 5)
        break; // sai do while com i valendo 5
    System.out.println(i);
}
System.out.println("Fim");
```

Ao executar o `break`, a execução do `while` para completamente. Temos a seguinte saída:

```
2
3
4
Fim
```

Vamos comparar com o `continue`:

```
int i = 1;
while (i < 10) {
    i++;
    if (i == 5)
```

```
        continue; // vai para a condição com o i valendo 5
    System.out.println(i);
}
```

Neste caso, iremos parar a execução da iteração apenas quando o valor da variável for igual a 5. Ao encontrar um `continue`, o código volta ao início da iteração, ao ponto do loop. Nossa saída agora é a seguinte:

```
2
3
4
6
7
8
9
10
Fim
```

Isto é, o `break` quebra o laço atual, enquanto o `continue` vai para a próxima iteração do laço.

Tome cuidado, pois um laço que tenha um `while` infinito do tipo `true` e que contenha um *break* é compilável, já que o compilador não sabe se o código poderá parar, possivelmente sim:

```
while(true) {
    if(1==2) break;
    System.out.println("em loop infinito compilável");
}
```

Os controladores de laços, `break` e `continue` podem ser aplicados no `for`. O `break` se comporta da mesma maneira que no `while` e no `do .. while`, parar o laço por completo. Já o `continue` faz com que a iteração atual seja abortada, executando em seguida a parte de *atualização* do `for`, e em seguida a de *condição*. Vamos ver o exemplo a seguir:

```
for (int i = 1; i < 10; i++) {
    if (i == 8) {
        break; // sai do for sem executar mais nada do laço.
    }
```

```
    if (i == 5) {
        // pula para a atualização sem executar o resto do corpo.
        continue;
    }
    System.out.println(i);
}
```

A saída desse código é :

```
1
2
3
4
6
7
```

Rótulos em laços (labeled loops)

Às vezes, encontramos a necessidade de "encaixar" um laço dentro de outro. Por exemplo, um `for` dentro de um `while` ou de outro `for`. Nesses casos, pode ser preciso manipular melhor a execução dos laços encaixados com os controladores de laços, `break` e `continue`.

```
for (int i = 1; i < 10; i++) { //laço externo
    for (int j = 1; j < 10; j++) { // laço interno
        if (i * j == 25)    {
            break; // qual for será quebrado?
        }
    }
}
```

Quando utilizamos o `break` ou o `continue` em laços encaixados, eles são aplicados no laço mais próximo. Por exemplo, nesse código, o `break` irá "quebrar" o `for` mais interno. Se fosse preciso "quebrar" o `for` mais externo, como faríamos?

Labeled statements

Podemos adicionar *labels* (rótulos) a algumas estruturas de código, e usá-los posteriormente para referenciarmos essas estruturas. Para declarar um

label usamos um nome qualquer (mesma regra de nomes de variáveis etc.) seguido de dois pontos (`:`). Por exemplo, podemos dar um label para um `for` como o que segue:

```
externo: //label
for(int i=0; i<10;i++){
    //código
}
```

Podemos usar esses *labels* para referenciar para qual loop queremos que o `break` ou o `continue` seja executado:

```
externo: for (int i = 1; i < 10; i++) {
  interno: for (int j = 1; j < 10; j++) {
    if (i * j == 25)    {
        break externo; // quebrando o for externo
    }
    if (i * j == 16) {
        continue interno; // pulando um iteração do for interno
    }
  }
}
```

Label http

O código a seguir imprime os valores de 1 a 10. Mas como ele compila sendo que temos uma URI logo antes do laço `for`?

```
http://www.caelum.com.br
    for (int i = 1; i <= 10; i++) {
        System.out.println(i);
    }
}
```

Um rótulo ou label pode estar presente antes de um *statement* qualquer, mas só podemos utilizar um *statement* de `break` ou `continue` caso o rótulo esteja referenciando um `for`, `while` ou `switch`:

```
void rotuloEmQualquerLugar() {
    rotulo: System.out.println("oi");
}

void rotuloEmQualquerLugarComBreakNaoCompila() {
    rotulo: System.out.println("oi");
    if(1<10) continue rotulo; // erro de compilação
}
```

Cuidado, mesmo dentro de um `for` ou similar, o `continue` e o `break` só funcionarão se forem relativos a um label dentro do qual estão, e do tipo `for`, `do...while`, `switch` ou `while`. Vale lembrar que `switch` só aceita `break`.

```
void rotuloEmQualquerLugarComBreakNaoCompila() {
    rotulo: System.out.println("oi");
    for(int i=0;i<10;i++) {
        break rotulo; // não compila
    }
}
void rotuloEmOutroLaco() {
    rotulo:
    for(int i=0;i<10;i++) {
        System.out.println("oi");
    }
    for(int i=0;i<10;i++) {
        break rotulo; // não compila
    }
}
```

Rótulos podem ser repetidos desde que não exista conflito de escopo:

```
void rotulosRepetidos() {
    rotulo: for (int i = 0; i < 10; i++) {
        break rotulo;
    }
    rotulo: for (int i = 0; i < 10; i++) {
        break rotulo;
    }
```

```
}
void rotulosRepetidosNestedNaoCompila() {
    rotulo: for (int i = 0; i < 10; i++) {
        rotulo: for (int j = 0; j < 10; j++) {
            break rotulo;
        }
    }
}
```

Não há conflito de nome entre rótulos e variáveis, pois seu uso é bem distinto. O compilador sabe se você está referenciando um rótulo ou uma variável:

```
class A {
    int rotulo = 15;
    void rotulosENomesDeVariaveisNaoConflitam() {
        rotulo: for (int i = 0; i < 10; i++) {
            int rotulo = 10;
            break rotulo;
        }
    }
}
```

Um mesmo statement pode ter dois labels:

```
void rotulosNoMesmoStatement() {
    primeiro: segundo: for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
}
```

Tome bastante cuidado com `breaks` e `continues` que são de `switch` mas parecem ser de `fors`:

```
class TestaLacos {
    public static void main(String[] args) {

        for(int i = 0; i < 4; i++) {
            System.out.println("Estou antes do switch");
            mario:
```

```
                guilherme: switch(i) {
                    case 0:
                    case 1:
                        System.out.println("Caso " + i);
                        for(int j = 0; j < 3; j++) {
                            System.out.println(j);
                            if(j==1) break mario;
                        }
                    case 2:
                        System.out.println("Estou em i = " + i);
                        continue;
                    case 3:
                        System.out.println("Cheguei no 3");
                        break;
                    default:
                        System.out.println("Estranho...");
                        break;
                }
                System.out.println("Estou apos o switch");
            }
        }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          fora: for(int a=0;a<30;a++)
4          for(int b=0;b<1;b++)
5          if(a+b==25) continue fora;
6          else if(a+b==20) break fora;
7          if(a==0) break fora;
8          else System.out.println(a);
9      }
10 }
```

a) Não compila.

b) Compila e imprime 1 até 29

c) Compila e não imprime nada

d) Compila e imprime 1 até 19, 21 até 24, 26 até 29

e) Compila e imprime 1 até 24, 26 até 29

f) Compila e imprime 1 até 19

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3          fora: for(int a=0;a<30;a++)
4          for(int b=0;b<1;b++)
5          if(a+b==25) continue fora;
6          else if(a+b==20) break;
7          else System.out.println(a);
8     }
9 }
```

a) Não compila.

b) Compila e imprime 0 até 29.

c) Compila e não imprime nada.

d) Compila e imprime 0 até 19, 21 até 24, 26 até 29.

e) Compila e imprime 0 até 24, 26 até 29.

f) Compila e imprime 0 até 19.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         int a = args.length;
4         int i = 0;
5         switch(a) {
6             case 0:
7             case 1:
8             for(i=0;i<15;i++, System.out.println(i))
9             if(i==5) continue;
```

```
10                if(i==15) break;
11                case 2:
12                System.out.println("2");
13            }
14            System.out.println("fim");
15        }
16 }
```

a) Não compila.

b) Compila e ao rodar com o argumentos imprime 0 até 14, 2, fim.

c) Compila e ao rodar com o argumentos imprime 1 até 15, 2, fim.

d) Compila e ao rodar com o argumentos imprime 0 até 4, 6 até 14, 2, fim.

e) Compila e ao rodar com o argumentos imprime 1 até 4, 6 até 15, fim.

f) Compila e ao rodar com o argumentos imprime 0 até 4, 6 até 9, 2, fim.

g) Compila e ao rodar com o argumentos imprime 1 até 4, 6 até 9, fim.

h) Compila e ao rodar com o argumentos imprime 1 até 4, 6 até 15, 2, fim.

i) Compila e ao rodar com o argumentos imprime 1 até 15, 2, fim.

j) Compila e ao rodar com o argumentos imprime 1 até 15, fim.

Capítulo 8

Trabalhando com métodos e encapsulamento

8.1 Crie métodos com argumentos e valores de retorno

Classes, *enums* e interfaces podem ter métodos definidos em seus corpos.

Todo método tem uma *assinatura* (também chamada de *interface*) e um *corpo* (somente no caso de métodos não abstratos).

A assinatura do método sempre tem:

- um nome seguindo as regras de identificadores;
- um tipo de retorno;

- um conjunto de parâmetros (pode ser vazio), cada um com seu nome e seu tipo;
- um modificador de visibilidade (nem que seja implícito, *package-private*).

E, ainda na assinatura, podemos ter:

- `final` em caso de herança, o método não pode ser sobrescrito nas classes filhas;
- `abstract` obriga as classes filhas a implementarem o método. O método abstrato **não** pode ter corpo definido;
- `static` atributos acessados direto na classe, sem instâncias;
- `synchronized` *lock* da instância;
- `native` não cai nesta prova. Permite a implementação do método em código nativo (*JNI*);
- `strictfp` não cai nesta prova. Ativa o modo de portabilidade matemática para contas de ponto flutuante.
- `throws <EXCEPTIONS>` após a lista de parâmetros, podemos indicar quantas exceptions quisermos para o `throws`.

A ordem dos elementos na assinatura dos métodos é sempre a seguinte, sendo que os modificadores podem aparecer em qualquer ordem: `<MODIFICADORES> <TIPO_RETORNO> <NOME> (<PARÂMETROS>) <THROWS_EXCEPTIONS>`

Parâmetros

Em Java, usamos parâmetros em métodos e construtores. Definimos uma lista de parâmetros sempre declarando seus tipos e nomes e separando por vírgula:

```
class Param {
    void teste(int a, int b) {

    }
}

// chamada
p.teste(1, 2);
```

A declaração das variáveis é feita na declaração dos métodos. A inicialização dos valores é feita por quem chama o método. (Note que, em Java, não é possível ter valores `default` para parâmetros e todos são obrigatórios, não podemos deixar de passar nenhum).

O único modificador possível de ser marcado em parâmetros é `final`, para indicar que aquele parâmetro não pode ter seu valor modificado depois da chamada do método (considerado boa prática):

```
class Param {
    void teste (final int a) {
        a = 10; // não compila
    }
}
```

Promoção em parâmetros

Temos que saber que nossos parâmetros também estão sujeitos à promoção de primitivos e ao polimorfismo. Por exemplo, a classe a seguir ilustra as duas situações:

```
class Param {
    void primitivo (double d) {

    }

    void referencia (Object o) {

    }
}
```

O primeiro método espera um `double`. Mas se chamarmos passando um `int`, um `float` ou qualquer outro tipo compatível, este será promovido a `double` e a chamada funciona:

```
Param p = new Param();
p.primitivo(10);
p.primitivo(10L);
p.primitivo(10F);
p.primitivo((short) 10);
p.primitivo((byte) 10);
p.primitivo('Z');
```

A mesma coisa ocorre com o método que recebe `Object`: podemos passar qualquer um que **é um** `Object`, ou seja, qualquer objeto:

```
Param p = new Param();
p.referencia(new Carro());
p.referencia(new Moto());
```

Retornando valores

Todo método pode retornar um valor ou ser definido como `void`, quando não devolve nada:

```
class A {
    int numero() {
        return 5;
    }
    void nada() {
        return;
    }
}
```

No caso de métodos de tipo de retorno `void` (nada), podemos omitir a última instrução:

```
class A {
    void nada() {
        // return; // pois esta linha é opcional
    }
}
```

Um método desse tipo também pode ter um retorno antecipado:

```
class A {
    void nada(int i) {
        if(i >= 0) return;
        System.out.println("negativo");
    }
}
```

Não podemos ter nenhum código que seria executado após um retorno:

```
class A {
    void nada(int i) {
        if(i >= 0) {
            return;

            // não compila, pois nunca chegará aqui
            System.out.println("era positivo ou zero");
        }
        System.out.println("negativo");
    }
}
```

Todo método que possui um tipo de retorno definido (isto é, diferente de `void`), deve retornar algo ou jogar uma `Exception` em cada um dos caminhos de saída possíveis do método, caso contrário o código não compila:

```
String metodo(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    //não compila, o que acontece se não for nem if nem else if?
}
```

Lembre-se que isso é feito pelo compilador, então ele não sabe os valores da variável `a` e se todos os casos foram cobertos:

```
String metodo(int a) {
    if(a > 0) {
```

```
        return "positivo";
    } else if(a <= 0) {
        return "negativo ou zero";
    }
    //não compila, o que acontece se não for nem if nem else if?
    //o compilador não consegue analisar os dois casos
}
```

Podemos jogar uma `exception` ou colocar um `return`:

```
String metodo(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    return "zero";
}

String metodo2(int a) {
    if(a > 0) {
        return "positivo";
    } else if(a < 0) {
        return "negativo";
    }
    throw new RuntimeException("não quero zero!");
}
```

Métodos que não retornam nada não podem ter seu resultado atribuído a uma variável:

```
void metodo() {
    System.out.println("oi");
}
void metodo2() {
    // não compila, o método acima não retorna nada
    int i = metodo();
}
```

Pelo outro lado, mesmo que um método retorne algo, seu retorno pode ser ignorado:

```
int metodo() {
    System.out.println("oi");
    return 5;
}
void metodo2() {
    int i = metodo(); // i = 5
    // chamei novamente e não retornei nada, sem problemas
    metodo();
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          x(args.length);
4      }
5      static int x(final int l) {
6          for(int i=0;i<100;i++) {
7              switch(i) {
8                  case l:
9                      System.out.println(l);
10                     if(l==i) return;
11                 case 0:
12                     System.out.println(0);
13             }
14         }
15         System.out.println("Fim");
16         return -1;
17     }
18 }
```

a) Não compila.

b) Compila e ao rodar com cinco parâmetros, imprime 0, 5 e Fim.

c) Compila e ao rodar com cinco parâmetros, imprime 0, 5, -1 e Fim.

d) Compila e ao rodar com cinco parâmetros, imprime 0 e 5.

e) Compila e ao rodar com cinco parâmetros, imprime 0, 5 e -1.

f) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0 e Fim.

g) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0, -1 e Fim.

h) Compila e ao rodar com cinco parâmetros, imprime 0 e 5.

i) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0 e -1.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        x(args.length);
    }
    static int x(final int l) {
        for(int i=0;i<100;i++) {
            switch(i) {
                case l:
                    System.out.println(l);
                    if(l==i) return 3;
                case 0:
                    System.out.println(0);
            }
        }
        System.out.println("Fim");
        return -1;
    }
}
```

a) Não compila.

b) Compila e ao rodar com cinco parâmetros, imprime 0, 5 e Fim.

c) Compila e ao rodar com cinco parâmetros, imprime 0, 5, -1 e Fim.

d) Compila e ao rodar com cinco parâmetros, imprime 0 e 5.

e) Compila e ao rodar com cinco parâmetros, imprime 0, 5 e -1.

f) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0 e Fim.

g) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0, -1 e Fim.

h) Compila e ao rodar com cinco parâmetros, imprime 0 e 5.

i) Compila e ao rodar com cinco parâmetros, imprime 0, 5, 0 e -1.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(a(args.length));
    }
    static int a(int l) {
        if(l<10) return b(l);
        else return c();
    }
    static int b(int l) {
        if(l<10) return b(l);
        else return c();
    }
    static long c() {
        return 3;
    }
}
```

a) Não compila: erro ao invocar o método `b`.

b) Não compila: erro ao invocar o método `c`.

c) Não compila por um motivo não listado.

d) Compila e, ao chamar com 15 argumentos, imprime 3.

e) Compila e, ao chamar com 15 argumentos, entra em loop infinito.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(a(args.length)[0]);
    }
    static int,int a(int l) {
        if(l==0) return {0, 1};
        else return {1, 0};
    }
}
```

a) Não compila.

b) Ao invocar com nenhum parâmetro, imprime 0.

c) Ao invocar com 5 parâmetros, imprime 0.

8.2 Aplique a palavra chave static a métodos e campos

O modificador estático diz que determinado atributo ou método pertence à classe, e não a cada objeto. Com isso, você não precisa de uma instância para acessar o atributo, basta o nome da classe.

```
public class Carro {
    public static int totalDeCarros;
}
```

E depois, para acessar:

```
Carro.totalDeCarros = 5;
```

Um método estático é um método da classe, podendo ser chamado sem uma instância:

```
public class Carro{
    private static int totalDeCarros;


    public static int getTotalDeCarros() [
        return totalDeCarros;
    }
}

int i = Carro.getTotalDeCarros();
```

O que não podemos fazer é usar um método/atributo de instância de dentro de um método estático:

```
public class Carro{
    private static int totalDeCarros;
    private int peso;
```

```
    public static int getPeso() {
        return peso;
    }
}
```

Esse código não compila, pois peso é um atributo de instância. Se alguém chamar esse método, que valor ele retornaria, já que não estamos trabalhando com nenhuma instância de carro em específico?

Repare que a variável estática pode acessar um método estático, e esse método acessar algo ainda não definido e ter um resultado inesperado à primeira vista:

```
static int b = getMetodo();
public static int getMetodo() {
    return a;
}
static int a = 15;
```

O valor de `b` será 0, e não 15, uma vez que a variável `a` ainda não foi inicializada e possui seu valor padrão quando da execução do método `getMetodo`.

Outro caso interessante é que uma variável estática pode acessar outra estática, desde que a outra tenha um valor atribuído antes da definição da atual:

```
static int inicial = 10;
static int segunda = inicial + 5; // compila

static int outra;
static void inicializa() {
    outra = 10;
}
static int naoCompila = outra + 1;
// não compila, o método inicializa é ignorável
```

Um detalhe importante é que membros estáticos podem ser acessados através de instâncias da classe (além do acesso direto pelo nome da classe).

```
Carro c = new Carro();
int i = c.getTotalDeCarros();
```

Cuidado com essa sintaxe, que pode levar a acreditar que é um método de instância. É uma sintaxe estranha mas que compila e acessa o método estático normalmente.

Além disso, esteja atento pois, caso uma classe possua um método estático, ela não pode possuir outro método não estático com assinatura que a sobrescreveria (mesmo que em classe mãe/filha):

```
class A {
    static void a() { // não compila
    }
    void a() { // não compila
    }
}

class B {
    static void a() {
    }
}
class C extends B {
    void a() { // não compila
    }
}
```

Outro ponto importante a tomar nota é que o *binding* do método é feito em compilação, portanto, o método invocado não é detectado em tempo de execução. Leve em consideração:

```
class A {
    static void metodo() {
        System.out.println("a");
    }
}

class B extends A {
    static void metodo() {
        System.out.println("b");
    }
}
```

Caso o tipo referenciado de uma variável seja `A` em tempo de compilação, o método será o da classe `A`. Se for referenciado como `B`, será o método da classe `B`:

```
A a= new A();
        a.metodo(); // a

        B b= new B();
        b.metodo(); // b

        A a2 = b;
        a2.metodo(); // a
    }
}
```

A definição de uma variável estática pode invocar métodos e variáveis estáticas:

```
class A {
    static int idade = calculaIdade();
    static int calculaIdade() {
        return 18;
    }
}
```

A palavra-chave `static` pode ser aplicada a classes aninhadas, mas este tópico não é cobrado nesta primeira certificação.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         x();
4     }
5     static x() {
6         System.out.println("x");
7         y();
8     }
9     static y() {
```

```
10          System.out.println("y");
11      }
12  }
```

a) Não compila.

b) Imprime x, y.

c) Imprime y, x.

d) Imprime x.

e) Imprime y.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          x();
4      }
5      static void x() {
6          System.out.println("x");
7          y();
8      }
9      static void y() {
10         System.out.println("y");
11     }
12 }
```

a) Não compila.

b) Imprime x, y.

c) Imprime y, x.

d) Imprime x.

e) Imprime y.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B {
2      void y() {
```

```
3           this.z();
4       }
5       static void z() {
6           System.out.println("z");
7       }
8   }
9   class A {
10      public static void main(String[] args) {
11          new A().x();
12      }
13      static void x() {
14          new B().y();
15      }
16  }
```

a) Não compila ao tentar invocar `y`.

b) Não compila ao tentar invocar `z`.

c) Não compila ao tentar invocar `x`.

d) Compila e imprime `z`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1   class B{
2       static void x() {
3           System.out.println("x");
4       }
5       static void y() {
6           System.out.println("y");
7       }
8   }
9   class A extends B {
10      public static void main(String[] args) {
11          this.x();
12          A.y();
13      }
14  }
```

a) Não compila, erro ao invocar `x`.

b) Imprime `x`, `y`.

c) Não compila, erro ao invocar `y`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B{
2      static void x() {
3          System.out.println("x");
4      }
5      static void y() {
6          System.out.println("y");
7      }
8  }
9  class A extends B {
10     public static void main(String[] args) {
11         x();
12         A.y();
13     }
14 }
```

a) Não compila, erro ao invocar `x`.

b) Imprime `x`, `y`.

c) Não compila, erro ao invocar `y`.

8.3 Crie métodos sobrecarregados

Um método pode ter o mesmo nome que outro, desde que a chamada não fique ambígua: os **argumentos** que são recebidos têm de ser obrigatoriamente diferentes, seja em quantidade ou em tipos.

```
class Teste {
    public void metodo(int i) {
    }

    protected void metodo(double x) {
    }
}
```

Já o código a seguir não compila:

```
class Teste {
    public int metodo() {}
    protected double metodo() {}
}
```

Nesse exemplo, temos ambiguidade porque o tipo de retorno não é suficiente para distinguir os métodos durante a chamada.

O Java decide qual das assinaturas de método sobrecarregado (*overloaded*) será utilizada em **tempo de compilação**.

Métodos sobrecarregados podem ter ou não um retorno diferente e uma visibilidade diferente. Mas eles não podem ter exatamente os mesmos tipos e quantidade de parâmetros. Nesse caso, seria uma sobrescrita de método.

No caso de sobrecarga com tipos que possuem polimorfismo, como em `Object` ou `String`, o compilador sempre invoca o método com o tipo mais específico (menos genérico):

```
public class Teste {
    void metodo(Object o) {
        System.out.println("object");
    }
    void metodo(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Teste().metodo("string"); // imprime string
    }
}
```

Se quisermos forçar a invocação ao método mais genérico, devemos fazer o casting forçado:

```
public class Teste {
    void metodo(Object o) {
        System.out.println("object");
    }
```

```
    void metodo(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Teste().metodo((Object)"string"); // imprime object
    }
}
```

Um exemplo clássico é a troca de ordem, que é vista como sobrecarga, afinal são dois métodos totalmente distintos:

```
void metodo(String i, double x) {
}
void metodo(double x, String i) {
}
```

Porém, apesar de compiláveis, às vezes o compilador não sabe qual método deverá chamar. No caso a seguir, os números 2 e 3 podem ser considerados tanto `int` quanto `double`, portanto, o compilador fica perdido em qual dos dois métodos invocar, e decide não compilar:

```
public class Teste {
    void metodo(int i, double x) {
    }
    void metodo(double x, int i) {
    }

    public static void main(String[] args) {
        new Teste().metodo(2, 3);
    }
}
```

Isso também ocorre com referências, que é diferente do caso com tipo mais específico. Aqui não há tipo mais específico, pois onde um é mais específico, o outro é mais genérico:

```
public class Xpto {
    void metodo(Object o, String s) {
```

```
        System.out.println("object");
    }
    void metodo(String s, Object o) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Xpto().metodo("string", "string");
    }
}
```

Diferente do caso em que o segundo método é mais específico:

```
class Xpto2 {
    void metodo(Object o, Object o2) {
        System.out.println("object");
    }
    void metodo(String s, String s2) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Xpto2().metodo("string", "string"); // imprime string
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int x = b(15);
4          System.out.println(x);
5          System.out.println(15);
6          System.out.println(15.0);
7      }
8      static int b(int i) { return i; }
9      static double b(int i) { return i; }
10 }
```

a) Não compila.

b) Compila e imprime `15, 15, 15`.

c) Compila e imprime `15, 15, 15.0`.

d) Compila e imprime `15, 15.0, 15.0`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x = b(15);
        System.out.println(x);
        System.out.println(15);
        System.out.println(15.0);
    }
    static int b(int i) { return i; }
    static double b(double i) { return i; }
}
```

a) Não compila.

b) Compila e imprime `15, 15, 15`.

c) Compila e imprime `15, 15, 15.0`.

d) Compila e imprime `15, 15.0, 15.0`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println("[]");
    }
    public static void main(String... args) {
        System.out.println("...");
    }
}
```

a) Não compila.

b) Compila e imprime "[]".

c) Compila e imprime "...".

d) Compila e dá exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    int a(D d) { return 1; }
    int a(C c) { return 2; }
    int a(B b) { return 3; }
    int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    int a(D d) { return 1; }
    static int a(C c) { return 2; }
    static int a(B b) { return 3; }
    static int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    static int a(D d) { return 1; }
    static int a(C c) { return 2; }
    static int a(B b) { return 3; }
    static int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    static int a(D d, B b) { return 1; }
    static int a(C c, C c) { return 2; }
```

```
7       static int a(B b, B b) { return 3; }
8       static int a(A a, A a) { return 4; }
9       public static void main(String[] args) {
10          System.out.println(a(new D(), new D()));
11      }
12  }
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B{}
2  class C{}
3  class D extends B{}
4  class A {
5      static int a(D d, B b2) { return 1; }
6      static int a(C c, C c2) { return 2; }
7      static int a(B b, B b2) { return 3; }
8      static int a(A a, A a2) { return 4; }
9      public static void main(String[] args) {
10         System.out.println(a(new D(), new D()));
11     }
12 }
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

8.4 Diferencia entre o construtor padrão e construtores definidos pelo usuário

Quando não escrevemos um construtor na nossa classe, o compilador nos dá um construtor padrão. Esse construtor, chamado de *default* não recebe argumentos, tem a mesma visibilidade da classe e tem a chamada a `super()`.

A classe a seguir:

```
class A {
}
```

... na verdade, acaba sendo:

```
class A {
    A() {
        super();
    }
}
```

Caso você adicione um construtor qualquer, o construtor `default` deixa de existir:

```
class A {}
class B {
    B(String s) {}
}
class Teste {
    public static void main(String[] args) {
        new A(); // construtor padrão, compila
        new B(); // não existe mais construtor padrão
        new B("CDC"); // construtor existente
    }
}
```

Dentro de um construtor você pode acessar e atribuir valores aos atributos, suas variáveis membro:

```
class Teste {
    int i;
```

```
    Teste() {
        i = 15; // agora i vale 15
        System.out.println(i); // 15
    }

    public static void main(String[] args) {
        new Teste();
    }
}
```

Os valores inicializados com a declaração das variáveis são inicializados **antes** do construtor, justamente por isso o valor inicial de `i` é 0, o valor padrão de uma variável `int` membro:

```
class Teste {
    int i;
    Teste() {
        System.out.println(i); // vale 0 por padrão
        i = 15; // agora i vale 15
        System.out.println(i); // 15
    }

    public static void main(String[] args) {
        new Teste();
    }
}
```

Vale lembrar que variáveis membro são inicializadas automaticamente para: numéricas 0, `boolean false`, referências `null`.

Cuidado ao acessar métodos cujas variáveis ainda não foram inicializadas no construtor. O exemplo a seguir mostra um caso em que o método de inicialização é invocado antes de setar o valor da variável no construtor, o que causa um `NullPointerException`.

```
class A {

    int i = 15;
    String nome;
    int tamanho = tamanhoDoNome();
```

```
    A(String nome) {
        this.nome = nome;
    }

    int tamanhoDoNome() {
        return nome.length();
    }

    A() {
    }

}
```

Mesmo que inicializemos a variável fora do construtor, após a chamada do método pode ocorrer um erro, como no caso a seguir, de um outro `NullPointerException`:

```
class A {

    int i = 15;
    String nome;
    int tamanho = tamanhoDoSobrenome();
    String sobrenome = "Silveira";

    A(String nome) {
        this.nome = nome;
    }

    int tamanhoDoSobrenome() {
        return sobrenome.length();
    }

    A() {
    }

}
```

Mudar a ordem da declaração das variáveis resolve o problema, uma vez que o método é agora invocado após a inicialização da variável `sobrenome`:

```
class A {

    int i = 15;
    String nome;
    String sobrenome = "Silveira";
    int tamanho = tamanhoDoSobrenome();

    A(String nome) {
        this.nome = nome;
    }

    int tamanhoDoSobrenome() {
        return sobrenome.length();
    }

    A() {
    }

}
```

Cuidado ao invocar métodos no construtor e variáveis estarem nulas:

```
class Teste {
    String nome;
    Teste() {
        testaTamanho(); // NullPointerException
        nome = "aprendendo";
    }

    private void testaTamanho() {
        System.out.println(nome.length());
    }

    public static void main(String[] args) {
        new Teste();
    }
}
```

E mais cuidado ainda caso isso ocorra por causa de sobrescrita de método, em que também poderemos ter essa `Exception`:

```
class Base {
    String nome;
    Base() {
        testa();
        nome = "aprendendo";
    }

    void testa() {
        System.out.println("testa");
    }

}
class Teste extends Base {
    void testa() {
        System.out.println(nome.length());
    }
    public static void main(String[] args) {
        new Teste();
    }
}
```

Já se o método `testa` for privado, como o *binding* da chamada ao método é feito em compilação, o método invocado pelo construtor é o da classe mãe, sem dar a `Exception`:

```
class Base {
    String nome;
    Base() {
        testa();
        nome = "aprendendo";
    }

    private void testa() {
        System.out.println("testa");
    }

}
class Teste extends Base {
    void testa() {
```

```
        System.out.println(nome.length());
    }
    public static void main(String[] args) {
        new Teste();
    }
}
```

Você pode entrar em loop infinito, cuidado, `StackOverflow`:

```
class Teste {
    Teste() {
        new Teste();
    }
    public static void main(String[] args) {
        new Teste();
    }
}
```

Construtores podem ser de todos os tipos de modificadores de acesso: `private`, `protected`, `default` e `public`.

É comum criar um construtor privado e um método estático para criar seu objeto:

```
class Teste {
    private Teste() {
    }

    public static Teste cria() {
        return new Teste();
    }
}
```

Tenha muito cuidado com um método com nome do construtor. Se colocar um `void` na frente, vira um método:

```
class Teste {
  void Teste() {
    System.out.println("Construindo");
  }
```

```
  public static void main(String[] args) {

    new Teste();
    // não imprime nada, definimos um método e não o construtor
    new Teste().Teste();
    // agora imprime Construindo
  }
}
```

Existem também blocos de inicialização que não são cobrados na prova.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      final String n;
3      A() {
4          a();
5          n = "aprendendo";
6      }
7      void a() {
8          System.out.println("testa");
9      }
10 }
11 class B extends A {
12     void a() {
13         System.out.println(n.length());
14     }
15     public static void main(String[] args) {
16         new B();
17     }
18 }
```

a) Não compila.

b) Compila e imprime "testa".

c) Compila e imprime `length`.

d) Compila e dá exception.

e) Compila e não imprime nada.

8.5 Crie e sobrecarregue construtores

Construtores também podem ser sobrecarregados:

```
class Teste {
    public Teste() {
    }
    public Teste(int i) {
    }
}
```

Cuidado com os exemplos de sobrecarga com `varargs`, como vimos antes, e no caso de herança.

Quando existem dois construtores na mesma classe, um construtor pode chamar o outro através da chamada `this`. Note que loops não compilam:

```
class Teste {
    public Teste() {
        System.out.println("construtor simples");
    }
    public Teste(int i) {
        this();
    }
    public Teste(String s) {
        this(s, s);  // não compila, loop
    }
    public Teste(String s, String s2) {
        this(s); // não compila, loop
    }
}
```

Temos que tomar cuidado com sobrecarga da mesma maneira que tomamos cuidado com sobrecarga de métodos: os construtores invocados seguem as mesmas regras que as de métodos.

Quando um método utiliza `varargs`, se ele possui uma variação do método sem nenhum argumento e invocarmos sem argumento, ele chamará o método sem argumentos (para manter compatibilidade com versões anteriores do Java):

```
void desativa(Cliente... clientes) {
    System.out.println("varargs");
}
void desativa() {
    System.out.println("sem argumento");
}
void metodo() {
    desativa(); // imprime sem argumento
}
```

A instrução `this` do construtor deve ser sempre a primeira dentro do construtor:

```
class Teste {
    Teste() {
        String valor = "valor...";
        this(valor); // não compila
    }

    Teste(String s) {
        System.out.println(s);
    }

    public static void main(String[] args) {
        new Teste();
    }
}
```

Justo por isso não é possível ter duas chamadas a `this`:

```
class Teste {
    Teste() {
        this(valor);
        this(valor); // não compila
    }

    Teste(String s) {
        System.out.println(s);
    }
```

```
    public static void main(String[] args) {
        new Teste();
    }
}
```

A instrução `this` pode envolver instruções:

```
class Teste {
    Teste() {
        this(valor());
    }

    private static String valor() {
        return "valor...";
    }

    Teste(String s) {
        System.out.println(s);
    }

    public static void main(String[] args) {
        new Teste();
    }
}
```

A instrução não pode ser um método da própria classe, pois o objeto não foi construído ainda:

```
class Teste {
    Teste() {
        this(valor()); // valor não é estático, não compila
    }

    private String valor() {
        return "valor...";
    }

    Teste(String s) {
        System.out.println(s);
    }
```

```
    public static void main(String[] args) {
        new Teste();
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { B() { this(1); } B(int i) { this(); } }
2 class A {
3     public static void main(String[] args) {
4         new B();
5     }
6 }
```

a) Não compila.

b) Compila e joga exception.

c) Compila e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B() { B(A a) {} B() {} }
2 class C() { C(B b) {} C() {} }
3 class A {
4     public static void main(String[] args) {
5         new A(); new B(); new C();
6     }
7 }
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { B(A a) {} B() {} }
2 class C { C(B b) {} C() {} }
3 class A {
4     public static void main(String[] args) {
5         new A(); new B(); new C();
6     }
7 }
```

a) Não compila ao invocar o construtor de A.

b) Não compila ao invocar o construtor de B.

c) Não compila ao invocar o construtor de C.

d) Não compila na definição das classes B e C.

e) Compila e joga exception.

f) Compila e não imprime nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { B(A a) {} B() {} }
2 class C { C(B b) {} C() {} }
3 class A {
4     public static void main(String[] args) {
5         new C(new B(new A()));
6     }
7 }
```

a) Não compila ao invocar o construtor de A.

b) Não compila ao invocar o construtor de B.

c) Não compila ao invocar o construtor de C.

d) Não compila na definição das classes B e C.

e) Compila e joga exception.

f) Compila e não imprime nada.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { B(A a) {new C(); } B() { new C(this);} }
2 class C { C(B b) {new B(new A());} C() {new B();} }
3 class A {
4     public static void main(String[] args) {
5         new C(new B(new A()));
6     }
7 }
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

8.6 Aplique modificadores de acesso

Os modificadores de acesso, ou modificadores de visibilidade, servem para definir quais partes de cada classe (ou se uma classe inteira) estão visíveis para serem utilizadas por outras classes do sistema. Só é permitido usar um único modificador de acesso por vez:

```
private public int x; // não compila
```

O Java possui os seguintes modificadores de acesso:

- `public`
- `protected`
- Nenhum modificador, chamado de `default`
- `private`

Classes e interfaces só aceitam os modificadores `public` ou `default`.

Membros (construtores, métodos e variáveis) podem receber qualquer um dos quatro modificadores.

Variáveis locais (declaradas dentro do corpo de um método ou construtor) e parâmetros não podem receber nenhum modificador de acesso, mas podem receber outros modificadores.

Top Level Classes e Inner Classes

Classes internas (*nested classes* ou *inner classes*) são classes que são declaradas dentro de outras classes. Esse tipo de classe pode receber qualquer modificador de acesso, já que são consideradas membros da classe onde foram declaradas (*top level class*).

Nesta certificação não são cobradas classes internas, apenas *top level classes*.

Para entender como os modificadores funcionam, vamos imaginar as seguintes classes:

```
1 package forma;
2
3 class Forma{
4     double lado;
5     double getArea(){
6         return 0;
7     }
8 }
```

```
1 package forma;
2
3 class Quadrado extends Forma{}
```

```
1 package forma.outro;
2 import forma.*;
3
4 class Triangulo extends Forma{}
```

Public

O modificador `public` é o menos restritivo de todos. Classes, interfaces e membros marcados com esse modificador podem ser acessados de qualquer componente, em qualquer pacote. Vamos alterar nossa classe `Forma`, marcando-a e todos seus membros com o modificador `public`:

```
package forma;

public class Forma{
    public double lado;
    public double getArea(){
        return 0;
    }
}
```

Agora vamos fazer um teste:

```
package forma.outro;
import forma.*;

public class TesteOutroPacote{

    public static void main(String... args){
        Forma f = new Forma(); //acesso a classe forma
        f.lado = 5.5; //acesso ao atributo lado
        f.getArea(); //acesso ao método getArea()
    }
}
```

Repare que, mesmo nossa classe `TesteOutroPacote` estando em um pacote diferente da classe `Forma`, é possível acessar a classe e todos os membros declarados como `public`.

Protected

Membros definidos com o modificador `protected` podem ser acessados por classes e interfaces no mesmo pacote, e por qualquer classe que estenda aquela onde o membro foi definido, independente do pacote.

Vamos modificar nossa classe `Forma` para entendermos melhor:

```
package forma;

public class Forma{
    protected double lado; // agora protected
    public double getArea(){}
}
```

Com o modificador `protected`, nossa classe de testes em outro pacote não compila mais:

```
package forma.outro;
import forma.*;

public class TesteOutroPacote{

    public static void main(String... args){
        Forma f = new Forma();
        f.lado = 5.5; // erro de compilação
        f.getArea();
    }
}
```

Se criarmos uma nova classe de teste no pacote `forma`, conseguimos acessar novamente o atributo:

```
1 package forma;
2
3 public class Teste{
4
5     public static void main(String... args){
6         Forma f = new Forma();
7         f.lado = 5.5; // compila normal, mesmo pacote
8     }
9 }
```

Embora esteja em um pacote diferente, a classe `Triangulo` consegue acessar o atributo `lado`, já que ela estende da classe `Forma`:

```
package forma.outro;
import forma.*;
```

```
class Triangulo extends Forma{

    public void imprimeLado(){
        //Como é uma classe filha, acessa
        //normalmente os membros protected da classe mãe.
        System.out.println("O Lado é " + lado);
    }
}
```

Agora repare que, se efetuarmos o casting do objeto atual para uma `Forma`, não podemos acessar seu lado:

```
package outro;
import forma.*;

class Triangulo extends Forma{

    public void imprimeLado(){
        // compila
        System.out.println("O Lado é " + lado);

        // não compila
        System.out.println("O Lado é " + ((Forma) this).lado);
    }
}
```

Isso ocorre porque estamos dizendo que queremos acessar a variável membro `lado` de um objeto através de uma referência para este objeto, e não diretamente. Diretamente seria o uso puro do `this` ou nada. Nesse caso, após usar o `this`, usamos um casting, o que deixa o compilador perdido.

Default

Se não definirmos explicitamente qual o modificador de acesso, podemos dizer que aquele membro está usando o modificador `default`, também chamado de `package private`. Neste caso, os membros da classe só serão visíveis dentro do mesmo pacote:

```
package forma;

public class Forma{
    protected double lado;
    public double getArea(){
        return 0;
    }
    double getPerimetro(){ //default access
        return 0;
    }
}
```

O método `getPerimetro()` só será visível para todas as classes do pacote `forma`. Nem mesmo a classe `Triangulo` que, apesar de herdar de `Forma`, está em outro pacote consegue ver o método.

```
package outro;
import forma.*;

class Triangulo extends Forma{

    public void imprimePerimetro(){
        //Erro de compilação na linha abaixo
        System.out.println("O Perímetro é " + getPerimetro());
    }
}
```

Palavra-chave default

Lembre-se! A palavra-chave `default` é usada para definir a opção padrão em um bloco `switch`, ou para definir um valor inicial em uma *Annotation*. Usá-la em uma declaração de classe ou membro é inválido e causa um erro de compilação:

```
default class Bola{ //ERRO
    default String cor; // ERRO
}
```

A partir do Java 8, a palavra `default` também pode ser usada para definir uma implementação inicial de um método.

Mas e se declararmos uma classe com o modificador `default`? Isso vai fazer com que aquela classe só seja visível dentro do pacote onde foi declarada. Não importa quais modificadores os membros dessa classe tenham, se a própria classe não é visível fora de seu pacote, nenhum de seus membros é visível também.

Veja a classe `Quadrado`, que está definida com o modificador `default`:

```
package forma;

class Quadrado extends Forma{}
```

Veja o seguinte código, usando a classe `TesteOutroPacote`. Perceba que não é possível usar a classe `Quadrado`, mesmo importando todas as classes do pacote `forma`:

```
package outro;
import forma.*;

public class TesteOutroPacote{

    public static void main(String... args){
        Quadrado q = new Quadrado(); // erro, esta classe não é
                                     //visível
```

```
    }
}
```

Linha com erro de compilação

Eventualmente, na prova, é perguntado em quais linhas ocorreram os erros de compilação. É bem importante prestar atenção nesse detalhe.

Por exemplo, neste caso, o erro sempre acontecerá quando tentarmos acessar a classe `Quadrado`, que não é visível fora de seu pacote:

```
package outro;

//import de todas as classes PÚBLICAS do pacote, nenhum erro
import forma.*;


public class TesteOutroPacote{

    public static void main(String... args){
        // erro na linha 8, Quadrado não é visível, pois não
        // é pública
        Quadrado q = new Quadrado();
    }
}
```

O mesmo código pode apresentar erro em uma linha diferente, apenas mudando o `import`. Repare que o código a seguir dá erro nas duas linhas, tanto do `import` quanto na tentativa de uso:

```
package outro;
// erro na linha 3, não podemos importar classes não públicas
import forma.Quadrado;

public class TesteOutroPacote{

    public static void main(String... args){

        //Erro, pois Quadrado não é acessível.
        Quadrado q = new Quadrado();
    }
}
```

É muito importante testar vários trechos de código, para ver exatamente em quais linhas de código o erro de compilação aparecerá.

Private

`private` é o mais restritivo de todos os modificadores de acesso. Membros definidos como `private` só podem ser acessados de dentro da classe e de nenhum outro lugar, independente de pacote ou herança:

```
package forma;

public class Forma{
    protected double lado;
    public double getArea(){}

    //cor só pode ser acessada dentro da classe Forma,
    //nem as classe Quadrado e Triangulo conseguem acessar
    private String cor;

}
```

> **Private e classes aninhadas ou anônimas**
>
> Classes aninhadas ou anônimas podem acessar membros privados da classe onde estão contidas. Na certificação tais classes não são cobradas.

Métodos privados e padrão não podem ser sobrescritos. Se uma classe o "sobrescreve", ele simplesmente é um método novo, portanto não podemos dizer que é sobrescrita. Veremos isso mais a fundo na seção sobre sobrescrita. 9.1

Resumo das regras de visibilidade

Todos os membros da classe com o modificador de `private` só podem ser acessados de dentro dela mesma.

Todos os membros da classe sem nenhum modificador de visibilidade, ou seja, com visibilidade **package-private**, podem ser acessados de dentro da própria classe ou de dentro de qualquer outra classe, interface ou enum do mesmo pacote.

Todos os membros da classe com o modificador `protected` podem ser acessados:

- de dentro da classe, ou de dentro de qualquer outra classe, interface ou enum do mesmo pacote;
- de dentro de alguma classe que deriva direta ou indiretamente da classe, independente do pacote. O membro `protected` só pode ser chamado através da referência `this`, ou por uma referência que seja dessa classe filha.

Todos os membros da classe com o modificador `public` podem ser acessados de qualquer lugar da aplicação.

E não podemos ter classes/interfaces/enums top-level como `private` ou `protected`.

Uma classe é dita *top-level* se ela não foi definida dentro de outra classe, interface ou enum. Analogamente, são definidas as interfaces top-level e os enums top-level.

1) Escolha a opção adequada ao tentar compilar e rodar o `Teste`. Arquivo no diretório atual:

```
1 import modelo.Cliente;
2 class Teste {
3     public static void main(String[] args) {
4         new Cliente("guilherme").imprime();
5     }
6 }
```

Arquivo no diretório `modelo`:

```
1 package modelo;
2
3 public class Cliente {
4     private String nome;
5     Cliente(String nome) {
6         this.nome = nome;
7     }
```

```
8       public void imprime() {
9           System.out.println(nome);
10      }
11  }
```

a) Não compila: erro na classe `Teste`.

b) Não compila: erro na classe `Cliente`.

c) Erro de execução: método `main`.

d) Roda e imprime "Guilherme".

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      private static int a(int b) {
3          return b(b)-1;
4      }
5      private static int b(int b) {
6          return b-1;
7      }
8      public static void main(String[] args) {
9          System.out.println(new A().a(5));
10     }
11 }
```

a) Não compila nas invocações de métodos.

b) Não compila na declaração de variáveis e métodos.

c) Compila e imprime 3.

d) Compila e dá erro.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      private public int a(int b) {
3          return b(b)-1;
4      }
5      private static int b(int b) {
```

```
6           return b-1;
7       }
8       public static void main(String[] args) {
9           System.out.println(new A().a(5));
10      }
11  }
```

a) Não compila nas invocações de métodos.

b) Não compila na declaração de variáveis e métodos.

c) Compila e imprime 3.

d) Compila e dá erro.

4) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
1 package a;
2 import b.*;
3 public class A extends B { protected int a(String s)
4                            {return 2;} }
```

```
1 package b;
2 import a.*;
3 public class B { public int a(Object s) {return 1;} }
```

```
1 import a.*;
2 import b.*;
3 class A {
4     public static void main(String[] args) {
5         System.out.println(new A().a("a"));
6     }
7 }
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

5) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
1 package a;
2 import b.*;
3 public class A extends B { protected int a(String s)
4                            {return 2;} }
```

```
1 package b;
2 import a.*;
3 public class B { public int a(Object s) {return 1;} }
```

```
1 import a.*;
2 import b.*;
3 class C {
4     public static void main(String[] args) {
5         System.out.println(new A().a("a"));
6     }
7 }
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

6) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
1 package a;
2 import b.*;
3 public class A extends B { protected int a(String s)
4                            {return 2;} }
```

```
1 package b;
2 import a.*;
3 public class B { default int a(Object s) {return 1;} }
```

```
import a.*;
import b.*;
class C {
    public static void main(String[] args) {
        System.out.println(new A().a("a"));
    }
}
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    static int bs=0;
    final int b = ++bs;
    private B() {}
    static B b() { return new B(); }
}
class A {
    public static void main(String[] args) {
        System.out.println(B.b().b);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 0.

d) Compila e dá erro de execução.

8.7 Aplique princípios de encapsulamento a uma classe

A **assinatura** de um método é o que realmente deve importar para o usuário de alguma classe. Segundo os bons princípios do encapsulamento, a implementação dos métodos deve estar *encapsulada* e não deve fazer diferença para o usuário.

O que é importante em uma classe é **o que ela faz** e não **como ela faz**. *O que ela faz* é definido pelos comportamentos expostos, ou seja, pelos métodos e suas assinaturas.

O conjunto de assinaturas de métodos visíveis de uma classe é chamado de **interface de uso**. É através dessas operações que os usuários vão se comunicar com os objetos dessa classe.

Mantendo os detalhes de implementação de nossas classes "escondidos", evitamos que mudanças na forma de implementar uma lógica quebre vários pontos de nossa aplicação.

Uma das formas mais simples de começar a encapsular o comportamento de uma classe é escondendo seus atributos. Podemos fazer isso facilmente usando a palavra-chave `private`:

```
public class Pessoa{
    private String nome;
}
```

Caso precisemos acessar um desses atributos a partir de outra classes, teremos que criar um método para liberar o acesso de leitura desse atributo. Seguindo a especificação dos *javabeans*, esse método seria um *getter*. Da mesma forma , se precisarmos liberar a escrita de algum atributo, criamos um método *setter* :

```
public class Pessoa{
    private String nome;

    public String getNome() {
        return nome;
    }
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Com essa abordagem, poderíamos fazer uma validação em nossos métodos, para evitar que nossos atributos fiquem com estado inválido. Por exemplo, podemos verificar se o nome possui pelo menos 3 caracteres:

```
public class Pessoa{
    private String nome;
    private String sobrenome;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        if(nome!= null && nome.trim().length() >= 3)
            this.nome = nome;
        else{
            throw new IllegalArgumentException(
             "Nome deve possuir " + "pelo menos 3 caracteres");
        }
    }
}
```

Encapsulamento é muito mais do que atributos privados e *getters* e *setters*. Não é nosso foco aqui discutir boas práticas de programação, e sim o conhecimento necessário para passar na prova. Em questões sobre encapsulamento sempre, fique atento à alternativa que esconde mais detalhes de implementação da classe analisada. A prova pode utilizar tanto o termo *encapsulation* como *information hiding* para falar sobre encapsulamento (ou esconder informações).

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     private int b;
3     public int getB() { return b; }
```

```
4     public void setB(int b) { this.b= b; }
5 }
6 class A {
7     public static void main(String[] args) {
8         new B().setB(5);
9         System.out.println(new B().getB());
10    }
11 }
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     private int b;
3     public int getB() { return b; }
4     public void setB(int b) { this.b= b; }
5 }
6 class A {
7     public static void main(String[] args) {
8         B b = new B();
9         b.setB(5);
10        System.out.println(b.getB());
11    }
12 }
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     private int b;
3     public int getB() { return b; }
```

```
4     public void setB(int b) { b= b; }
5 }
6 class A {
7     public static void main(String[] args) {
8         B b = new B();
9         b.setB(5);
10        System.out.println(b.getB());
11    }
12 }
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     int b;
3     public void setB(int b) { b= b; }
4 }
5 class A {
6     public static void main(String[] args) {
7         B b = new B();
8         b.setB(5);
9         System.out.println(b.b);
10    }
11 }
```

a) Não compila, pois não é possível ter `setter` sem `getter`.

b) Compila e imprime 0.

c) Compila e imprime 5.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B{
2     private final int b;
3     B(int b) { this.b = b;}
```

```
4       public int getB() { return b; }
5       public void setB(int b) { b= b; }
6   }
7   class A {
8       public static void main(String[] args) {
9           B b = new B(10);
10          b.setB(5);
11          System.out.println(b.getB());
12      }
13  }
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

d) Compila e imprime 10.

8.8 Determine o efeito que ocorre com referências a objetos e a tipos primitivos quando são passados a outros métodos e seus valores mudam

As informações que queremos enviar para um método devem ser passadas como parâmetro. O domínio de como funciona a passagem de parâmetro é fundamental para a prova de certificação.

O requisito para entender passagem de parâmetro no Java é saber como funciona a pilha de execução e o *heap* de objetos.

A pilha de execução é o "lugar" onde são empilhados os métodos invocados na mesma ordem em que foram chamados.

O *heap* é o "lugar" onde são guardados os objetos criados durante a execução.

Considere o exemplo a seguir:

```
class Teste {
    public static void main(String[] args) {
```

```
        int i = 2;
        teste(i);
    }

    private static void teste(int i) {
        for (int j = 0; j < i; j++) {
            new String("j = " + j);
        }
    }
}
```

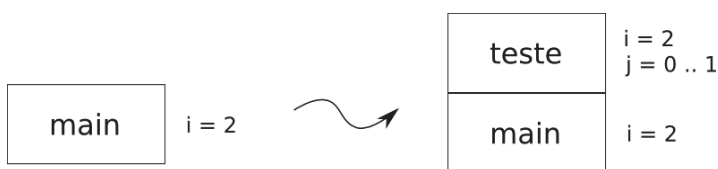



Figura 8.1: Pilha de execução

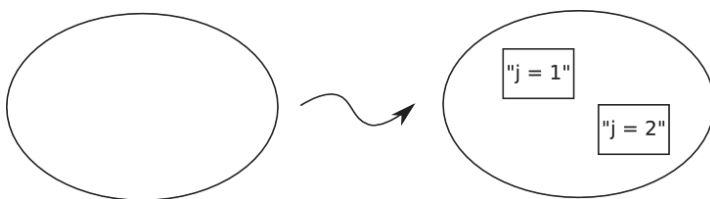



Figura 8.2: Heap

A passagem de parâmetros é feita por cópia de valores. Dessa forma, mudanças nos valores das variáveis definidas na lista de parâmetros de um método não afetam variáveis de outros métodos.

Passagem de parâmetros primitivos

Veja o seguinte código:

```
class Teste {
    public static void main(String[] args) {
        int i = 2;
        teste(i);
        System.out.println(i);
    }

    static void teste(int i) {
        i = 3;
    }
}
```

Ao executar a classe `Teste`, será impresso o valor `2`. É necessário perceber que as duas variáveis com o nome `i` estão em métodos diferentes. Há um `i` no `main()` e outro `i` no `teste()`. Alterações em uma das variáveis não afetam o valor da outra.

Passagem de parâmetros de referência

Agora veja esta classe:

```
class Teste {
    public static void main(String[] args) {
        Prova prova = new Prova();
        prova.tempo = 100;
        teste(prova);
        System.out.println(prova.tempo);
    }

    static void teste(Prova prova) {
        prova.tempo = 210;
    }
}

class Prova {
    double tempo;
}
```

Esse exemplo é bem interessante e causa muita confusão. O que será impresso na saída, ao executar a classe `Teste`, é o valor `210`. Os dois métodos

têm variáveis com o mesmo nome (`prova`). Essas variáveis são realmente independentes, ou seja, mudar o valor de uma não afeta o valor da outra.

Por outro lado, como são variáveis não primitivas, elas guardam referências e, neste caso, são referências que apontam para o mesmo objeto. Modificações nesse objeto podem ser executadas através de ambas as referências.

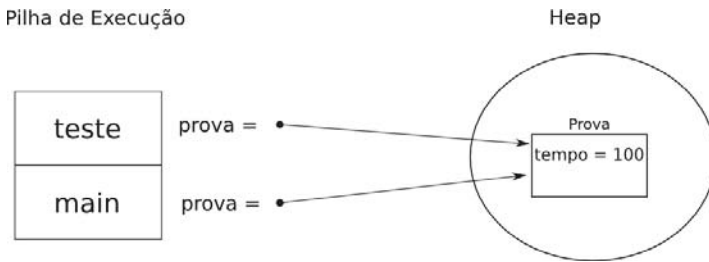



Figura 8.3: Passagem de parâmetros não primitivos

Mas se eu trocar a referência, só estou trocando nesta variável local, e não no objeto referenciado, como no exemplo do `teste2`, em que estamos trocando somente a referência local e não o outro:

```
class Prova {
    int tempo;
}
class TestaReferenciaEPrimitivo {
    public static void main(String[] args) {
        Prova prova = new Prova();
        prova.tempo = 100;
        teste(prova);
        System.out.println(prova.tempo);

        teste2(prova);
        System.out.println(prova.tempo);


        int i = 2;
        i = teste(i);
        System.out.println(i);
```

```
    }
    static void teste2(Prova prova) {
        prova = new Prova();
        prova.tempo = 520;
    }

    static void teste(Prova prova) {
        prova.tempo = 210;
    }

    static int teste(int i) {
        i = 5;
        System.out.println(i);
        return i;
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class A {
2      public static void main(String[] args) {
3          int i = 150;
4          i = ++s(i);
5          System.out.println(i);
6      }
7      static int s(int i) {
8          return ++i;
9      }
10 }
```

a) Não compila.

b) Compila e imprime 150.

c) Compila e imprime 151.

d) Compila e imprime 152.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```java
class A {
    public static void main(String[] args) {
        int[] i = {150, 151};
        i = s(i);
        System.out.println(i[1]);
    }
    static int[] s(int[] i) {
        int[] j = {i[0], i[1]};
        i[1]++;
        return j;
    }
}
```

a) Não compila.

b) Compila e imprime 150.

c) Compila e imprime 151.

d) Compila e imprime 152.

e) Compila e imprime 153.

Capítulo 9

Trabalhando com herança

9.1 Implementando herança

Em Java, podemos usar herança simples entre classes com o `extends`. A nomenclatura usada é de **classe mãe** (*parent class*) e **classe filha** (*child class*), ou **superclasse** e **subclasse**.

Herança entre classes permite que um código seja reaproveitado, de maneira que a classe filha reutilize o código da parte mãe, preocupando-se principalmente em sua especialização. A filha especializa a classe mais genérica. Herança em Java pode ser entre classes, reaproveitando membros, ou herança de uma interface, com a qual reaproveitamos interfaces de métodos.

```
class Mae {
}
```

```
class Filha extends Mae {
}
class Neta extends Filha {
}
```

Vale lembrar que toda classe que não define de quem está herdando herda de `Object`:

```
class Explicito extends Object {
}
class Implicito {
    // extends Object por padrão
}
class FilhoDeImplicito extends Implicito{
    // também herda de Object, indiretamente
}
```

Mas não podemos herdar de duas classes:

```
class Simples1 {}
class Simples2 {}
class Complexa extends Simples1, Simples2 {
    // não compila
}
```

Para podermos herdar de uma classe, a classe mãe precisa ser visível pela classe filha e pelo menos um de seus construtores também:

```
class Pai {
    Pai(int x) {
    }
}
class Filho1 extends Pai{
    // não compila pois o construtor padrão chama super()
    // e o Pai não tem construtor vazio
}
class Filho2 extends Pai{
    Filho2() {
        super(15); //compila
    }
}
```

Além disso, a classe mãe não pode ser `final`:

```
final class Pai {
}
class Filho extends Pai {
    // não compila, Pai é final
}
class Mae {
}
final class Filha extends Mae {
    // uma classe final pode estender de alguém, compila
}
```

Herança de métodos e atributos

Todos os métodos e atributos de uma classe mãe são herdados (independente das visibilidades).

```
class X {
    int x;
    public void y() {
    }
}
class Y {
    // tenho um x, e o método y
}
```

Dependendo da visibilidade e das classes envolvidas, a classe filha não consegue enxergar o membro herdado. No exemplo a seguir, herdamos o atributo mas não o enxergamos diretamente.

```
class X {
    private int x;

    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
```

```
    }
}

class Y extends X {
    public void metodo () {
        this.x = 5; // não compila: "x has private access in X"

        this.setX(10); // compila e altero o x herdado mas
                       // não visível
    }
}
```

Métodos estáticos e herança

Não existe herança de métodos estáticos. Mas quando herdamos de uma classe com métodos estáticos, podemos chamar o método da classe mãe usando o nome da filha (embora não seja uma boa prática):

```
class W {
    static void metodo() {
    }
}

class  Z extends W {
}

class Teste {
    public static void main(String[] args) {
        Z.metodo(); // melhor seria escrever W.metodo()
    }
}

class W {
    public static void metodo() {
        System.out.println("w");
    }
}
class Z extends W {
    public static void metodo() {
```

```
        // não existe super em contexto estático, não compila
        super.metodo();

    }
}
```

Por não existir herança, o modificador `abstract` não é aceito em métodos estáticos.

Podemos até escrever na classe filha um método estático de mesmo nome, mas isso **não é sobrescrita** (alguns chamam de redefinição):

```
class W {
    public static void metodo() {
        System.out.println("w");
    }
}
class Z extends W {
    public static void metodo() {
        System.out.println("z");
    }
}
public class Teste {
    public static void main(String[] args) {
        System.out.println(W.metodo()); // w
        System.out.println(Z.metodo()); // z
    }
}
```

Na verdade você até segue as regras de sobrescrita de método (visibilidade e retorno), mas no polimorfismo ele não funciona como métodos normais. Ele simplesmente funciona com o tipo da variável em tempo de compilação e não o tipo do objeto em tempo de execução:

```
public class Teste {
    public static void main(String[] args) {
        W w = new W();
        w.metodo(); // w

        Z z = new Z();
```

```
        z.metodo(); // z

        W zPolimorfadoComoW = z;
        zPolimorfadoComoW.metodo();
        // este último imprime w,
        // pois o binding é feito em compilação:
        // zPolimorfadoComoW.metodo é uma referencia
        // em compilação para W
    }
}
```

Construtores e herança

Não existe herança de construtores. O que existe é a classe filha chamar o construtor da mãe.

Sobrescrita de atributos

Não existe sobrescrita de atributos. Podemos, sim, ter um atributo na classe filha com mesmo nome da mãe, mas não chamamos de sobrescrita. Nesses casos, o objeto vai ter 2 atributos diferentes, um da mãe (acessível com `super`) e um na filha (acessível com `this`).

Object

Em Java, toda classe é obrigada a usar a herança. Quando não escrevemos `extends` explicitamente, estamos herdando de `java.lang.Object` automaticamente.

Isso quer dizer que todo objeto em Java **é um** `Object` e, portanto, herda todos os métodos da classe `Object` (por isso esses são muito importantes).

Há vários métodos em `Object`, que veremos ao longo do curso, mas o mais simples talvez seja o `toString`, que podemos sobrescrever em nossas classes para devolver alguma `String` que represente o objeto:

```
class Carro {
    String cor;

    public String toString() {
        return "Um carro de cor " + this.cor;
    }
}
```

Temos que lembrar que o `toString` é chamado automaticamente para nós quando usamos o objeto no contexto de `String`:

```
Carro c = new Carro();
c.cor = "Verde";

System.out.println(c); // chama toString

String s = "Mensagem: " + c; // chama toString
System.out.println(s);
```

1)
```
class A {
    public void metodo(long l) {
    }
}
class B extends A{
    protected void metodo(int i) {
    }
}
```

Compila?

2)
```
import java.io*;
class Veiculo {
    protected void liga () throws IOException {}
}
class Carro extends Veiculo {
    public void liga() throws FileNotFoundException {}
}
```

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B extends C { int m(int a) { return 1; } }
2 class C extends A { int m(double b) { return 3; } }
3 class A extends B {
4     int m(String c) { return 3; }
5     public static void main(String[] args) {
6         System.out.println(new C().m(3));
7     }
8 }
```

a) O código não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

e) Compila e imprime 1, 2, 3.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { int m(int a) { return 1; } }
2 class C { int m(double b) { return 2; } }
3 class A extends B, C{
4     public static void main(String[] args) {
5         System.out.println(new C().m(3));
6     }
7 }
```

a) O código não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 1, 2.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { private B() {} static B B(String s)
2           { return new B(); } }
3 class A {
```

```
4     public static void main(String[] args) {
5         B b = B.B("t");
6     }
7 }
```

a) Não compila.

b) Compila e imprime "t".

c) Compila e não imprime nada.

d) Compila e joga uma exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B { private B() {} static B B(String s)
2           { return new B(); } }
3 class A extends B {
4     public static void main(String[] args) {
5         B b = B.B("t");
6     }
7 }
```

a) Não compila.

b) Compila e imprime "t".

c) Compila e não imprime nada.

d) Compila e joga uma exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     private String s;
3     protected B() {}
4     static A B(String s) {
5         return new A();
6     }
7 }
8 class A extends B {
9     A(String s) {
```

```
10          this.s = s;
11      }
12      public static void main(String[] args) {
13          B b = A.B("t");
14          System.out.println(b.s);
15      }
16  }
```

a) Não compila.

b) Compila e imprime "t".

c) Compila e não imprime nada.

d) Compila e joga uma exception.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B {
2      protected String s;
3      protected B() {}
4      static A B(String s) {
5          return new A();
6      }
7  }
8  class A extends B {
9      A(String s) {
10         this.s = s;
11     }
12     public static void main(String[] args) {
13         A b = A.B("t");
14         System.out.println(b.s);
15     }
16 }
```

a) Não compila.

b) Compila e imprime "t".

c) Compila e não imprime nada.

d) Compila e joga uma exception.

9.2 Desenvolva código que mostra o uso de polimorfismo

Reescrita ou sobrescrita é a maneira como uma subclasse pode *redefinir* o comportamento de um método que foi herdado de uma das suas superclasses (direta ou indiretamente).

```
class Veiculo {
    public void liga() {
        System.out.println("Veiculo está sendo ligado!");
    }
}

class Carro extends Veiculo {
    public void liga() {
        System.out.println("Carro está sendo ligado!");
    }
}
```

Agora considere:

```
public class Teste{
    public static void main(String [] args){
        Veiculo v = new Carro();
        v.liga();
    }
}
```

O método chamado aqui será o da classe `Carro`, independente de a referência ser do tipo `Veiculo` (o que importa é o objeto).

Qual método será executado é descoberto em **tempo de execução** (a *assinatura* é decidida em tempo de compilação!), isso é a chamada virtual de método (*virtual method invocation*).

Para reescrever um método, é necessário:

- exatamente o mesmo nome;
- os parâmetros têm que ser iguais em tipo e ordem (nomes podem mudar);

- retorno do método deve ser igual ou mais específico que o da mãe;
- visibilidade deve ser igual ou maior que o da mãe;
- exceptions lançadas devem ser iguais ou menos que na mãe;
- método na mãe não pode ser `final`.

Se essas regras não forem respeitadas, pode haver um erro de compilação, ou o método declarado não será considerado uma reescrita do método.

A regra sobre visibilidade é: um método reescrito só pode ter visibilidade maior ou igual à do método que está sendo reescrito. (Essa não é uma regra mágica! Faz todo o sentido; pense um pouco sobre o que poderia acontecer se essa regra não existisse).

O código a seguir não compila, pois `ligar` é público na classe mãe, então só pode ser reescrito com visibilidade pública:

```
class Veiculo {
    public void liga() {
        System.out.println("Veiculo esta sendo ligado!");
    }
}

class Carro extends Veiculo {
    protected void liga() {
        System.out.println("Carro esta sendo ligado!");
    }
}
```

Muito cuidado com interfaces, pois a definição de um método é, por padrão, `public` e o exercício pode apresentar uma pegadinha de compilação:

```
interface A {
    void a();
}
class B implements A {
    void a() {
        // não compila, o método deveria ser público
```

```
    }
}
class C implements A {
    public void a() {
        // compila
    }
}
```

Estranhamente, um método sobrescrito pode ser abstrato, dizendo para o compilador que quem herdar dessa classe terá que sobrescrever o método original:

```
class A {
    void a() {
    }
}
abstract class B extends A {
    abstract void a(); // sobrescrevendo como abstrato
}

class C extends B{
    // não compila, não redefiniu a
}

class D extends B{
    void a() {
        // compila pois redefiniu a
    }
}
```

Sobre o **retorno covariante**: permite que a classe filha tenha um retorno igual ou mais específico polimorficamente (um subtipo).

Cuidado! O retorno covariante não vale para tipos primitivos. Um exemplo de retorno covariante:

```
class A {
    List<String> metodo () {
        // devolve lista
    }
```

```
}

class B extends A {
    ArrayList<String> metodo() {
        // devolve array list
    }
}
```

Outra regra importante sobre reescrita é a assinatura em relação ao *lançamento de exceções* (`throws`). Um método reescrito só pode lançar as mesmas exceções *checked* ou menos que o métodos que está sendo reescrito (quanto às *unchecked*, não há regras e sempre podemos lançar quantas quisermos).

```
import java.sql.SQLException;
import java.io.IOException;

class A {
    public void metodo () throws SQLException, IOException {
    }
}

class B extends A {
    public void metodo () throws IOException {
    }
}
```

Esse código compila, pois o método na classe `A` lança menos exceções que na classe mãe, respeitando a regra. Já o código a seguir não compila:

```
import java.sql.SQLException;
import java.io.IOException;

class A {
    public void metodo () throws SQLException {
    }
}

class B extends A {
    public void metodo () throws IOException {
```

```
    }
}
```

Apesar de ambos os métodos lançarem apenas uma exceção, não é isso que importa, pois elas são diferentes. Outro caso que não compila:

```
import java.io.IOException;

class A {
    public void metodo () throws IOException {
    }
}

class B extends A {
    public void metodo () throws Exception {
    }
}
```

`Exception` é muito mais que `IOException`.

Repare que, quando dizemos *menos exceções que na mãe*, isso indica não apenas quantidade, mas também devemos considerar o polimorfismo. Se trocarmos o exemplo anterior, compilamos:

```
import java.io.IOException;

class A {
    public void metodo () throws Exception {
    }
}

class B extends A {
    public void metodo () throws IOException {
    }
}
```

Compila, pois `IOException` é mais específico que `Exception` na árvore de herança.

Polimorfismo e chamadas de métodos

Imagine as classes:

```
class Veiculo {
    void liga() {
        System.out.println("ligando o veiculo");
    }
}
class Carro extends Veiculo {
    void liga() {
        System.out.println("ligando o carro");
    }

    void desliga() {
    }
}
```

Se tivermos um objeto do tipo `Carro` com uma referência do tipo `Carro`, ou seja, sem usar polimorfismo, podemos fazer:

```
Carro c = new Carro();
c.liga(); // ligando o carro
c.desliga();
```

Conseguimos chamar os dois métodos. E, como estamos trabalhando com sobrescrita, o método `liga` chamado é o da classe filha `Carro`.

Mas e se usarmos polimorfismo e a referência para `Veiculo`?

```
Veiculo v = new Carro();
v.liga(); // ligando o carro?
v.desliga();
```

Vamos linha por linha: primeiro, podemos chamar um `Carro` de `Veiculo` porque ele *é um* (compila sem problemas). Podemos também chamar o método `liga` pois ambas as classes o possuem. Mas o método que será invocado será o da classe filha, o sobrescrito.

Já a chamada ao método `desliga` não compilará, porque ele não está definido na classe `Veiculo`. Como a referência é desse tipo, o método (que existe no objeto) não é visível.

A regra é: para saber se um método de um objeto pode ser chamado, olhamos para o tipo da referência em tempo de compilação. Para realmente chamar o método em tempo de execução, devemos olhar para o objeto ao que demos `new`.

Essa regra faz sentido quando pensamos em um método polimórfico como o seguinte:

```
void metodo (Veiculo v) {
    v.liga(); // compila
    v.desliga(); // não compila
}
```

Se passarmos um objeto `Carro` para o método, teoricamente ambas as chamadas funcionariam, já que a classe possui tanto o `liga` quanto o `desliga`.

Mas imagine uma classe `Moto` que não tem o método `desliga`. Como `Moto` *é um* `Veiculo`, podemos passar como argumento. O que aconteceria se pudéssemos ter chamado o `desliga` na referência `Veiculo`? Alguma coisa estaria errada.

Portanto, a regra geral é que somente podemos acessar os métodos de acordo com o tipo da referência, pois a verificação da existência do método é feita em compilação. Mas qual o método que será invocado, isso será conferido dinamicamente, em execução.

Um ponto muito importante é que o compilador **nunca** sabe o valor das variáveis depois da linha que as cria. Ou seja, o compilador não sabe se estamos passando um `Carro` ou uma `Moto`. O que ele sabe é apenas o tipo da variável; no caso, `Veiculo`. E como `Veiculo` não tem o método `desliga`, o código não pode compilar.

this, super e sobrescrita de métodos

Na ocasião em que um método foi sobrescrito, podemos utilizar as palavras-chave `super` e `this` para deixar explícito qual método desejamos invocar:

```
class A {
    public void metodo() {
```

```
        System.out.println("a");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo(); // imprime a
    }

    public void metodo2() {
        metodo(); // imprime b, a
        super.metodo(); // imprime a
    }
}
```

E se eu invocar o segundo método na primeira classe? Sem o `this`?

```
class A{
    public void metodo() {
        System.out.println("a");
        metodo2();
    }
    public void metodo2() {
        System.out.println("metodo 2 do pai");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo();
    }
    public void metodo2() {
        System.out.println("c");
        metodo();
        super.metodo();
    }
    public static void main(String[] args) {
        new B().metodo2();
    }
}
```

O Java entra em loop infinito, uma vez que o método será invocado no objeto. Então faremos o *lookup* do `metodo2` dinamicamente, encontrando o `metodo2` que chama `metodo`, que chama `metodo` do pai, que chama novamente `metodo2`. Note que o *lookup* dos métodos, o *binding* dos métodos, é feito em execução mesmo se invocarmos dentro de um próprio objeto. Até mesmo o uso da palavra-chave `this` não evitaria isso, causando o loop:

```
class A{
    public void metodo() {
        System.out.println("a");
        this.metodo2();
    }
    public void metodo2() {
        System.out.println("metodo 2 do pai");
    }
}
class B extends A {
    public void metodo() {
        System.out.println("b");
        super.metodo();
    }
    public void metodo2() {
        System.out.println("c");
        metodo();
        super.metodo();
    }
    public static void main(String[] args) {
        new B().metodo2();
    }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     void x() throws IOException {
3         System.out.println("c");
4     }
5 }
```

```
class C extends B {
    void x() throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) {
        new C().x();
    }
}
```

a) Não compila.

b) Compila e imprime ‘b‘.

c) Compila e imprime ‘c‘.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x() throws IOException {
        System.out.println("c");
    }
}
class C extends B {
    void x() throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) throws IOException {
        new C().x();
    }
}
```

a) Não compila.

b) Compila e imprime ‘b‘.

c) Compila e imprime ‘c‘.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(double i) throws IOException {
        System.out.println("c");
    }
}
class C extends B {
    void x(int i) throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) throws IOException {
        new C().x(3.2);
    }
}
```

a) Não compila.

b) Compila e imprime ‘b‘.

c) Compila e imprime ‘c‘.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.io.*;
2 class B {
3     void x(double i) throws IOException {
4         System.out.println("c");
5     }
6 }
7 class C {
8     void x(int i) throws FileNotFoundException {
9         System.out.println("b");
10    }
11 }
12 class A {
13     public static void main(String[] args) throws IOException {
14         new C().x(3.2);
15     }
16 }
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.io.*;
2 interface B {
3     public void x(double i) throws IOException {
4         System.out.println("c");
5     }
6 }
7 class C implements B {
8     public void x(int i) throws FileNotFoundException {
9         System.out.println("b");
10    }
11 }
```

```
12 class A {
13     public static void main(String[] args) throws IOException {
14         new C().x(3);
15     }
16 }
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 import java.io.*;
2 class B {
3     void x(int i) throws IOException {
4         System.out.println("c");
5     }
6 }
7 abstract class C extends B throws IOException {
8     abstract void x(int i);
9 }
10 abstract class D extends C {
11     void x(int i) throws IOException {
12         System.out.println("d");
13     }
14 }
15 class E extends D {
16 }
17 class A {
18     public static void main(String[] args) throws IOException {
19         new E().x(32);
20     }
21 }
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e imprime 'd'.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(i<0) return;
        x(-1);
        System.out.println("c");
    }
}
abstract class C extends B {
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e imprime 'd'.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(i<0) return;
        this.x(-1);
        System.out.println("c");
    }
}
abstract class C extends B {
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException  {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e imprime 'b','c'.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(i<0) return;
        super.x(-1);
        System.out.println("c");
    }
}
abstract class C extends B {
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime 'b'.

c) Compila e imprime 'c'.

d) Compila e imprime 'b','c'.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

9.3 Diferencie entre o tipo de uma referência e o tipo de um objeto

Sempre que estendemos alguma classe ou implementamos alguma interface, estamos relacionando nossa classe com a classe mãe ou interface usando um relacionamento chamado de **é um**.

Se `Carro extends Veiculo`, dizemos que *Carro **é um** Veiculo*. Ou se `ArrayList implements List` dizemos que *ArrayList **é um** List*.

O relacionamento de **é um** é um dos recursos mais poderosos da orientação a objetos. E é chamado formalmente de **polimorfismo**.

Polimorfismo é a capacidade que temos de referenciar um objeto de formas diferentes, segundo seus relacionamentos de *é um*.

Em especial, usamos polimorfismo quando escrevemos:

```
Veiculo v = new Carro();
List l = new ArrayList();
```

As heranças e implementações vão formando uma árvore que terá sempre como raiz a classe `Object`. Assim, direta ou indiretamente, todo objeto **é um** `Object`.

O polimorfismo pode ser aplicado à passagem de parâmetros (e é aí que está seu grande poder). Imagine as classes:

```
class Veiculo {}
class Carro extends Veiculo {}
```

```
class Moto extends Veiculo {}
class Onibus extends Veiculo {}
class Conversivel extends Carro {}
```

Se temos um método que recebe `Veiculo`, podemos passar qualquer um daqueles objetos:

```
void metodo (Veiculo v) {

}

// .....

metodo(new Carro());
metodo(new Moto());
metodo(new Onibus());
metodo(new Veiculo());
metodo(new Conversivel());
```

Dessa forma, conseguimos obter um forte reaproveitamento de código.

Repare que, quando usamos polimorfismo, estamos mudando o tipo da referência, mas nunca o tipo do objeto. Em Java, objetos nunca mudam seu tipo, que é aquele onde demos `new`. O que fazemos é chamar (referenciar) o objeto de várias formas diferentes. Chamar de várias formas.... é o polimorfismo.

Podemos referenciar um objeto pelo seu próprio tipo, por uma de suas classes pai, ou por qualquer interface implementada por ele, direta ou indiretamente:

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
public class Teste {
    public static void main(String[] args) {

        // mesmo tipo, compila
        D d = new D();
```

```
        // D extends C, todo D é um C, compila
        C c = new D();
        C c2 = d;

        // D implements B, todo D implementa B, compila
        B b = new D();
        B b2 = d;

        // D implements A indiretamente, compila
        A a = new D();
        A a2 = a;

        D d2 = new C(); // não, C não é D, não compila

        D d3 = new D();
        C c3 = d3; // compila
        D d4 = c3; // não compila, por mais que o ser humano
                   // saiba, em execução, nem todo C é um D.
    }
}
```

E como funciona o acesso às variáveis membro e aos métodos? Se temos uma referência para a classe mãe, não importa o que o valor seja em tempo de execução, o compilador não conhece o tempo de execução, então ele só compila chamadas aos métodos definidos na classe mãe:

```
class Veiculo {
    public void liga() { }
}
class Carro {
    public void mudaMarcha() {}
}

// teste
Veiculo v = new Veiculo();
v.liga(); // compila

Carro c = new Carro(); // ok
```

```
c.mudaMarcha(); // compila

Veiculo v2 = c;
v2.liga(); // todo veiculo tem método liga, compila
v2.mudaMarcha(); // não compila, nem todo veiculo tem
```

Mesmo em casos em que "achamos" que todo veículo tem, se o método não foi definido na classe de referência, o codigo não compila:

```
abstract class Veiculo {
    public void liga() { }
}
class Carro {
    public void desliga() { }
}
class Moto {
    public void desliga() { }
}

Carro c = new Carro(); // ok
c.desliga(); // compila

Veiculo v2 = c;
v2.desliga(); // não compila, Veiculo não tem o método desliga
              // definido
```

O mesmo valerá para variáveis membro:

```
class Veiculo {
    int velocidade;
}
class Carro {
    int marcha;
}

// teste
Veiculo v = new Veiculo();
v.velocidade = 3; // compila

Carro c = new Carro(); // ok
```

```
c.marcha = 1; // compila

Veiculo v2 = c;
v2.velocidade = 5; // compila
v2.marcha = 7; // não compila
```

E temos que cuidar de mais um caso específico: o que acontece se estamos trabalhando com pacotes distintos?

Se o método da classe pai que está sendo sobrescrito é `public`, os métodos que sobrescrevem devem ser `public`, então não tem muita graça.

Já se o método da classe pai é `protected`, os filhos são `protected` ou `public`, que também não tem graça, pois o filho mesmo em outro pacote já tinha acesso ao método do pai.

Mas o que acontece se o método no pai é `private` e eu tento sobrescrevê-lo? Ou se o método é `default` e tento sobrescrevê-lo em outro pacote? O mesmo valerá tanto para `private` quando para modificador de escopo padrão:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    void fecha() {
        System.out.println("fechando financeiro");
    }
}

package modelo;
public class Conta {
    void fecha() {
        System.out.println("fechando conta normal");
    }
}
```

Ao invocar o método `fecha` através de uma referência para `Conta` ou `ContaFinanceira`, o resultado será totalmente diferente:

```
ContaFinanceira c = new ContaFinanceira();
c.fecha();
Conta d = c;
d.fecha();
```

O código não compila, dependendo do pacote onde ele está. Como assim? Acontece que o método não foi sobrescrito, a classe filha nem sabe da existência do método (privado ou default) do pai, portanto o que ela fez foi criar um método totalmente novo.

Nesse caso, ao invocarmos o método durante compilação, o *binding* é feito para o método específico de cada uma delas, uma vez que são métodos totalmente diferentes. Se o código está no pacote de modelo, a chamada ao método `fecha` de `Conta` compila e imprimiria `normal`. Se estivermos no pacote `financeiro`, a chamada ao `ContaFinanceira` compila e imprime `financeiro`.

Lembre-se que os métodos privados terão um efeito equivalente: eles só são vistos internamente à classe onde foram definidos.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class D extends C {
2      void x() { System.out.println(1); }
3  }
4  class C extends B {
5      void x() { System.out.println(2); }
6  }
7  class B {
8      void x() { System.out.println(3); }
9      void y(B b) {
10         b.x();
11     }
12     void y(C b) {
13         c.x();
14     }
15     void y(D b) {
16         d.x();
17     }
18 }
19 class A {
20     public static void main(String[] args) {
21         new B().y(new C());
22     }
23 }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
    }
    void y(C c) {
        c.x();
    }
    void y(D d) {
        d.x();
    }
}
class A {
    public static void main(String[] args) {
        new B().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
    }
}
class A {
    public static void main(String[] args) {
        new B().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
    void y(C b) {
        x();
    }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
```

```
12      void y(B b) {
13          b.x();
14      }
15  }
16  class A {
17      public static void main(String[] args) {
18          new B().y(new C());
19      }
20  }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1   class D extends C {
2       void x() { System.out.println(1); }
3       void y(C b) {
4           x();
5       }
6   }
7   class C extends B {
8       void x() { System.out.println(2); }
9   }
10  class B {
11      void x() { System.out.println(3); }
12      void y(B b) {
13          b.x();
14      }
15  }
16  class A {
17      public static void main(String[] args) {
18          new D().y(new C());
19      }
20  }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    void fecha() {
        System.out.println("fechando financeiro");
    }
}
```

```
package modelo;
public class Conta {
    void fecha() {
        System.out.println("fechando conta normal");
    }
}
```

```
1 package codigo;
2 import modelo.*;
3 import financeiro.*;
4 class A {
5     public static void main(String[] args) {
6         new Conta().fecha();
7     }
8 }
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo 'fechando financeiro'.

d) Compila e roda, imprimindo 'fechando conta normal'.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    void fecha() {
        System.out.println("fechando financeiro");
    }
}
```

```
package modelo;
public class Conta {
    public void fecha() {
        System.out.println("fechando conta normal");
    }
}
```

```
1 package modelo;
2 import modelo.*;
3 import financeiro.*;
4 class A {
5     public static void main(String[] args) {
6         new Conta().fecha();
7     }
8 }
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo 'fechando financeiro'.

d) Compila e roda, imprimindo 'fechando conta normal'.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    void fecha() {
        System.out.println("fechando financeiro");
    }
}
```

```
package modelo;
public class Conta {
    protected void fecha() {
        System.out.println("fechando conta normal");
    }
}
```

```
1 package codigo;
2 import modelo.*;
3 import financeiro.*;
4 class A {
5     public static void main(String[] args) {
6         new Conta().fecha();
7     }
8 }
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo 'fechando financeiro'.

d) Compila e roda, imprimindo 'fechando conta normal'.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package financeiro;
public class ContaFinanceira extends modelo.Conta {
    public void fecha() {
        System.out.println("fechando financeiro");
    }
}
```

```
package modelo;
public class Conta {
    public void fecha() {
        System.out.println("fechando conta normal");
    }
}
```

```
1 package codigo;
2 class A {
3     public static void main(String[] args) {
4         new Conta().fecha();
5     }
6 }
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo 'fechando financeiro'.

d) Compila e roda, imprimindo 'fechando conta normal'.

10) O que acontece com o código a seguir?

```
interface Veiculo {
    int getMarcha();
    void liga();
}

abstract class Carro implements Veiculo {
    public void liga()  {
        System.out.println("ligado!");
    }
}

class CarroConcreto extends Carro implements Veiculo {
    public int getMarcha() {
        return 1;
    }
}
```

9.4 Determine quando é necessário fazer casting

Às vezes, temos referências de um tipo mas sabemos que lá há um objeto de outro tipo, um mais específico:

```
public class Teste{
    public static void main(String...args){
        Object[] objetos = new Object[100];

        String s = "certificacao";
        objetos[0] = s;

        String recuperada = objetos[0];
    }
}
```

O código acima não compila:

```
Teste.java:3: incompatible types
found   : java.lang.Object
required: java.lang.String
                String recuperada = objetos[0];
                                           ^
1 error
```

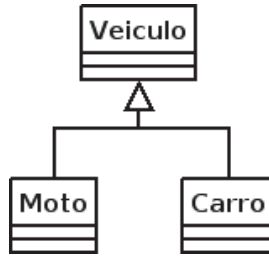
Temos um array de referências para `Object`. Nem todo `Object` é uma `String`, então o compilador não vai deixar você fazer essa conversão. Lembre-se que, em geral, o compilador não conhece os *valores* das variáveis, apenas seus tipos.

Vamos precisar *moldar* a referência para que o código compile:

```
String recuperada = (String) objetos[0];
```

A partir de agora, esse código compila. Mas será que roda? Durante a execução, o **casting** vai ver se aquele objeto é mesmo compatível com o tipo `String` (no nosso caso é). Se não fosse, ele lançaria uma `ClassCastException` (exceção *unchecked*).

Considere as classes:

```
class Veiculo {}
class Moto extends Veiculo {}
class Carro extends Veiculo {}
```

E o código:

```
Veiculo v = new Carro();
Moto m = v;
```

Na primeira linha, usamos polimorfismo para chamar um `Carro` de `Veiculo` (**é um**). Na segunda linha, o que o compilador sabe é que `v` é do tipo `Veiculo`. E *nem todo* `Veiculo` *é uma* `Moto`. Por isso, essa linha não compila.

Mas existem *alguns* `Veiculo` que são `Moto`. Então, o compilador deixa que façamos o casting:

```
Veiculo v = new Carro();
Moto m = (Moto) v;
```

Com isso, o código compila, mas repare que, em tempo de execução, `v` aponta para um objeto do tipo `Carro`. Quando o código for executado, haverá um erro de execução: `ClassCastException`. `Carro` **não** *é uma* `Moto`.

Cuidado que, se o *casting* for totalmente impossível, o compilador já acusará erro:

```
Carro c = new Carro();
Moto m = (Moto) c;
```

Um `Carro` **nunca** poderá ser uma `Moto`. Então, nem com casting isso compila.

É importante lembrar que quando não precisamos de casting, ele é opcional, portanto todas as linhas a seguir funcionam com ou sem casting:

```
String guilherme = "guilherme";
String nome = guilherme;
String nome2 = (String) guilherme;
Object nome3 = guilherme;
Object nome4 = (String) guilherme;
Object nome5 = (Object) guilherme;
```

REGRA GERAL!

Se você está subindo na hierarquia de classes, a autopromoção vai fazer tudo sozinho; e se você estiver descendo, vai precisar de casting. Se não houver um caminho possível, não compila nem com casting.

Na prova, faça sempre os diagramas de hierarquia de tipos que fica extremamente fácil resolver esses castings.

Casting com interfaces

Dado o código a seguir:

```
Carro c = new Carro();
Moto m = (Moto) c;
```

Quando dizemos que ele não compila, é porque um `Carro` nunca pode ser uma `Moto`. Mas como o compilador sabe que isso é impossível mesmo? Existe alguma chance de algum objeto de qualquer tipo ser, ao mesmo tempo, `Carro` e `Moto`?

```
class X extends Moto, Carro { // não compila!

}
```

A única maneira de isso acontecer seria se Java suportasse herança múltipla; aí escreveríamos uma classe que herdasse de `Carro` e `Moto` ao mesmo tempo. Como Java não tem herança múltipla, isso realmente é impossível de acontecer.

Mas e quando fazemos casting com interfaces envolvidas? Apesar de não existir herança múltipla, podemos implementar múltiplas interfaces! Fazer

casting para interfaces sempre é possível e vai compilar (há apenas uma exceção a essa regra).

Pegue uma interface *qualquer*, por exemplo `Runnable`. O código a seguir compila:

```
Carro c = new Carro();
Runnable r = (Runnable) c;
```

Um `Carro` *pode ser* um `Runnable`? Sabemos que a classe `Carro` propriamente não implementa essa interface. Mas existe a possibilidade de existir algum objeto em Java que seja, ao mesmo tempo, `Carro` e `Runnable`?

A resposta é sim! E se tivéssemos uma classe `CarroRodavel`?

```
class CarroRodavel extends Carro implements Runnable { ... }
```

O compilador não sabe o valor da variável `c` nesse exemplo. Ele não sabe que na verdade é uma instância de `Carro` e não de `CarroRodavel`. Ele sabe apenas que é do tipo `Carro` e, pela simples possibilidade de existir um objeto que seja `Carro` e `Runnable`, ele deixa o código compilar.

Mas repare que a classe `CarroRodavel` não existe no diagrama original. Mesmo assim, o código compila! Apenas com a *possibilidade* de existir uma classe dessa, o compilador já aceita aquele casting, mesmo que uma classe dessas não exista na prática.

Claro que o objeto é do tipo `Carro`, que não implementa `Runnable` e, em tempo de execução, vai ocorrer uma `ClassCastingException`.

E FINAL

Dizemos que o código anterior compila porque há a possibilidade de uma classe como `CarroRodavel` existir algum dia. Mas será que sempre há essa possibilidade mesmo?

Se a classe `Carro` for `final`, é impossível existir uma classe filha dela. E como a própria `Carro` não implementa `Runnable`, nesse caso, será impossível fazer o casting para `Runnable` (o próprio compilador já acusa erro).

> **Dica**
>
> Muitos exercícios são sobre casting de referência. Uma dica é seguir o que é possível, impossível e óbvio.
>
> Se é óbvio que o casting funciona, isso é, se a conversão é sempre verdade, a autopromoção faz sozinha.
>
> Se o casting é possível, mas nem sempre é verdade, o casting compila, mas pode lançar erro em tempo de execução.
>
> Se o casting é impossível, isto é, ele nunca pode dar certo, o código não vai compilar nem com casting.
>
> Em alguns livros, você encontra tabelas complicadas e grandes que o "ajudam" a decidir se o casting compila e roda, mas é muito mais fácil seguir pela lógica.

instanceof

O operador `instanceof` (`a instanceof Classe`) devolve `true` caso a referência `a` aponte para um objeto compatível (*assignable*, atribuível) ao tipo `Classe`.

```
Object c = new Carro();
boolean b1 = c instanceof Carro; // true
boolean b2 = c instanceof Moto; // false
```

O `instanceof` não compila se a referência em questão for obviamente incompatível, por exemplo:

```
String s = "a";
boolean b  = s instanceof java.util.List; // não compila
```

> **Detalhe**
>
> `instanceof` é um operador que deve ser usado com extremo cuidado no dia a dia. Em muitos casos, ele indica a fraca modelagem de um sistema, com blocos que parecem "*switchs*" e poderiam ser trocados por polimorfismo.

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {};
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        B b = new C();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
```

```
7 class E extends C {}
8 class A {
9     public static void main(String[] args) {
10        C c = (C) new B();
11    }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 interface Z {}
2 interface W {}
3 interface Y extends Z, W {}
4 class B {}
5 class C extends B implements Y {}
6 class D extends B implements Z, W {}
7 class E extends C {}
8 class A {
9     public static void main(String[] args) {
10        Y y = new D();
11    }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 interface Z {}
2 interface W {}
```

```
3 interface Y extends Z, W {}
4 class B {}
5 class C extends B implements Y {}
6 class D extends B implements Z, W {}
7 class E extends C {}
8 class A {
9     public static void main(String[] args) {
10        Y y = (Y) new D();
11    }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 interface Z {}
2 interface W {}
3 interface Y extends Z, W {}
4 class B {}
5 class C extends B implements Y {}
6 class D extends B implements Z, W {}
7 class E extends C {}
8 class A {
9     public static void main(String[] args) {
10        Z z = (Z) (B) new D();
11    }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        Y y = (Y) new A();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        D d = (D) (Y) (B) new D();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  interface Z {}
2  interface W {}
3  interface Y extends Z, W {}
4  class B {}
5  class C extends B implements Y {}
6  class D extends B implements Z, W {}
7  class E extends C {}
8  class A {
9    public static void main(String[] args) {
10     System.out.println(((B) (Z) (W) (Y) new D()) instanceOf D);
11   }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e imprime `true`.

d) Compila e imprime `false`.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  interface Z {}
2  interface W {}
3  interface Y extends Z, W {}
4  class B {}
5  class C extends B implements Y {}
6  class D extends B implements Z, W {}
7  class E extends C {}
8  class A {
9    public static void main(String[] args) {
10     System.out.println(((B) (Z) (W) (Y) new D()) instanceof D);
11   }
12 }
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e imprime `true`.

d) Compila e imprime `false`.

9.5 Use super e this para acessar objetos e construtores

Um construtor pode ser sobrecarregado assim como os métodos, e pode ter qualquer modificador de visibilidade.

O ponto mais importante sobre os construtores é que, para construir um objeto de uma classe filha, obrigatoriamente, precisamos chamar um construtor da classe mãe antes. Sempre, em todos os casos. Para chamar o construtor da mãe, usamos a chamada ao `super` (passando ou não argumentos):

```
class Mae {
    public Mae(String msg) {
        System.out.println(msg);
    }
}

class Filha extends Mae {
    public Filha(String nome) {
        super("construindo parte mae");
        System.out.println("construindo parte filha");
    }
}
```

Mas, na maioria dos casos, não chamamos o construtor da mãe explicitamente. Se nenhum construtor da mãe foi escolhido através da palavra `super(...)`, o compilador coloca **automaticamente** `super();` no começo do nosso construtor, sem nem olhar para a classe mãe.

```
class Mae {
    public Mae() {
        System.out.println("construindo parte mae");
    }
```

```
}

class Filha extends Mae {
    public Filha(String nome) {
        // super() esta implícito!!!
        System.out.println("construindo parte filha");
    }
}
```

Considerando agora o código:

```
public class X{
    public static void main(String [] args){
        Filha filha = new Filha("Teste");
    }
}
```

Vai primeiro imprimir "Construindo parte mae" e só depois "Construindo parte filha".

Uma outra possibilidade, no caso de termos mais de um construtor, é chamarmos outro construtor da própria classe através do `this()`:

```
class Mae {
    public Mae() {
        System.out.println("construindo parte mae");
    }
}

class Filha extends Mae {
    public Filha() {
        // super() implicito!
        System.out.println("construindo filha parte 1");
    }

    public Filha(String nome) {
        this();
        System.out.println("construindo filha parte 2");
    }
}
```

```
public class X{
    public static void main(String [] args){
        Filha filha = new Filha("Teste");
    }
}
```

Agora vai produzir "Construindo parte mae", "Construindo parte filha parte 1" e "Construindo parte filha parte 2".

Atenção, a chamada do construtor com `super` ou `this` só pode aparecer como primeira instrução do construtor. Portanto, só podemos fazer uma chamada desses tipos.

```
class Filha extends Object {
    public Filha() {
        // super() implicito!
    }

    public Filha(String nome) {
        this();
    }

    public Filha(int idade) {
        super();
        this(); // não compila, ou um ou outro!
    }
    public Filha(long valor) {
        this();
        this(); // não compila, só uma vez!
    }

    public Filha(char caracter) {
        super();
        super(); // não compila, só uma vez!
    }
}
```

this e variáveis membro

Por vezes, temos variáveis membro com o mesmo nome de variáveis locais. O acesso sempre será a variável local, exceto quando colocamos o `this`, que indica que a variável membro será acessada. O código a seguir imprimirá 3 e depois 5:

```
class Teste {
    int i = 5;
    void roda(int i) {
        System.out.println(i);
        System.out.println(this.i);
    }
    public static void main() {
        new Teste().roda(3);
    }
}
```

No acesso a variáveis membro com o `this` podem parecer que serão acessados somente valores da classe atual, mas buscam também nas classes da qual ela herda:

```
class A{
    int i = 5;
}
class Teste extends A{
    void roda(int i) {
        System.out.println(this.i); // imprime 5
    }
    public static void main() {
        new Teste().roda(3);
    }
}
```

Tentar acessar uma variável local com `this` não compila:

```
class Teste {
    void roda(int i) {
        System.out.println(this.i); // não há variável membro i
    }
```

```
    public static void main() {
        new Teste().roda(3);
    }
}
```

Como mostramos, caso a variável seja escondida por uma variável com mesmo nome em uma classe filha, podemos diferenciar o acesso à variável membro da classe filha ou da pai, explicitando `this` ou `super`:

```
class A{
    int i = 5;
}
class Teste extends A{
    int i = 10;
    void roda(int i) {
        System.out.println(i); // imprime 3
        System.out.println(this.i); // imprime 10
        System.out.println(super.i); // imprime 5
    }
    public static void main() {
        new Teste().roda(3);
    }
}
```

O `this` é em geral opcional para acessar um método do nosso objeto atual (se ele não foi redefinido, da classe mãe):

```
class A{
    int i() { return 5; }
}
class Teste extends A{
    void roda() {
        System.out.println(this.i()); // imprime 5
    }
    public static void main() {
        new Teste().roda();
    }
}
class Teste2 {
```

```
    int i() { return 5; }
    void roda() {
        System.out.println(this.i()); // imprime 5
    }
    public static void main() {
        new Teste().roda();
    }
}
```

this e super em variável membro

E o que acontece quando uma variável membro tem o mesmo nome que a definida na classe que herdamos? Se não definirmos o acesso através de `this` nem `super`, o acesso é à variável da classe filha. Se usarmos `this` é à classe filha novamente e se usarmos `super` é à classe pai:

```
class Veiculo {
    double velocidade = 30;
}
class Carro extends Veiculo {
    double velocidade = 50;
    void imprime() {
        System.out.println(velocidade); // 50
        System.out.println(this.velocidade); // 50
        System.out.println(super.velocidade); // 30
    }
}
class Teste {
    public static void main(String[] args) {
        Carro c = new Carro();
        c.imprime();
    }
}
```

Lembre-se que o binding de uma variável ao tipo é feito em compilação, portanto se tentarmos acessar a variável velocidade fora do `Carro` através de uma referência a `Carro`, o valor alterado é o da variável `Carro.velocidade`:

```
class Veiculo {
    double velocidade = 30;
}
class Carro extends Veiculo {
    double velocidade = 50;
    void imprime() {
        System.out.println(velocidade); // 1000
        System.out.println(this.velocidade); // 1000
        System.out.println(super.velocidade); // 30
    }
}
class Teste {
    public static void main(String[] args) {
        Carro c = new Carro();
        c.velocidade = 1000;
        c.imprime();
    }
}
```

E se fizermos o mesmo através de uma referência a `Veiculo`, alteramos a velocidade do `Veiculo`:

```
class Veiculo {
    double velocidade = 30;
}
class Carro extends Veiculo {
    double velocidade = 50;
    void imprime() {
        System.out.println(velocidade); // 50
        System.out.println(this.velocidade); // 50
        System.out.println(super.velocidade); // 1000
    }
}
class Teste {
    public static void main(String[] args) {
        Carro c = new Carro();
        ((Veiculo) c).velocidade = 1000;
        c.imprime();
    }
}
```

Estático não tem this nem super

Contextos estáticos não possuem nem `this` nem `super`, uma vez que o código não é executado dentro de um objeto:

```
class A{
    int i = 5;
}
class Teste extends A{
    int i = 10;
    public static void main() {
        this.i = 5; // this? não compila. código estático
        super.i = 10; // super? não compila. código estático
    }
}
```

Por fim, uma última restrição: interfaces não podem ter métodos estáticos, não compila (métodos `default` não são cobrados nesta prova).

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     int x = 1;
3 }
4 class A extends B {
5     static int x = 2;
6     public static void main(String[] args) {
7         System.out.println(x);
8     }
9 }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     int x = 1;
3 }
4 class A extends B {
5     static int x = 2;
6     public static void main(String[] args) {
7         System.out.println(this.x);
8     }
9 }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     int x = 1;
3 }
4 class A extends B {
5     static int x = 2;
6     public static void main(String[] args) {
7         System.out.println(super.x);
8     }
9 }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     int x = 1;
```

```
3 }
4 class A extends B {
5     static int x = 2;
6     public static void main(String[] args) {
7         System.out.println(new A().super.x);
8     }
9 }
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     void B() {
3     }
4     void B(String s) {
5         this();
6         this(s);
7     }
8 }
9 class A {
10     public static void main(String[] args) {
11         B b = new B();
12     }
13 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     B() {
3     }
4     B(String s) {
5         this();
6         this(s);
7     }
8 }
9 class A {
10     public static void main(String[] args) {
11         B b = new B();
12     }
13 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class B {
2     B() {
3     }
4     B(String s) {
5         this();
6     }
7 }
8 class A {
9     public static void main(String[] args) {
10         B b = new B();
11     }
12 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B {
2      B() {
3      }
4      B(String s) {
5          this();
6      }
7  }
8  class A {
9      public static void main(String[] args) {
10         String s = null;
11         B b = new B(s);
12     }
13 }
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B {
2      int x() { return y();}
3      int y() { return 3; }
4  }
5  class C extends B {
6      C() {
7          this(x());
8      }
9      C(int i) {
10         System.out.println(i);
11     }
12     int y() { return 2; }
```

```
13 }
14 class A {
15     public static void main(String[] args) {
16         new C();
17     }
18 }
```

a) Não compila.

b) Compila e imprime '2'.

c) Compila e imprime '3'.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  class B {
2      int x() { return y();}
3      int y() { return 3; }
4  }
5  class C extends B {
6      C() {
7          super();
8          z(x());
9      }
10     void z(int i) {
11         System.out.println(i);
12     }
13     int y() { return 2; }
14 }
15 class A {
16     public static void main(String[] args) {
17         new C();
18     }
19 }
```

a) Não compila.

b) Compila e imprime '2'.

c) Compila e imprime '3'.

9.6 Use classes abstratas e interfaces

Classes e métodos podem ser abstratos.

Uma classe abstrata pode não ter nenhum método abstrato:

```
// compila
abstract class SemMetodos {
}
```

Se uma classe tem um método que é abstrato, ela deve ser declarada como abstrata, ou não compilará.

```
// não compila, se tem método abstrato, tem que implementar
class ComMetodoAbstrato {
    public abstract void executa();
}
```

Uma classe abstrata não pode ser instanciada diretamente:

```
abstract class X{

}

public class Teste{
    public static void main(String[] args) {
        X x = new X();
    }
}
```

A classe `Teste` não compila! Classes abstratas não podem ser instanciadas diretamente:

```
Teste.java:7: X is abstract; cannot be instantiated
        X x = new X();
              ^
1 error
```

Um método abstrato é um método sem corpo, somente com a definição. Uma classe que tem um ou mais métodos abstratos precisa ser declarada como abstrata.

Não importa se o método foi escrito diretamente ou foi herdado:

```
abstract class Veiculo {
    public abstract void liga();
}

// compila pois implementou
class Moto extends Veiculo {
    public void liga() {

    }
}

// compila pois a classe é abstrata, com método herdado
// abstrato ainda
abstract class QuatroRodas extends Veiculo {
}

// não compila pois a classe não é abstrata,
// com método herdado abstrato ainda
class SemRodas extends Veiculo {
}
```

Um método abstrato tem de ser reescrito ou herdado pelas suas filhas concretas.

Agora veja o exemplo a seguir:

```
abstract class Veiculo {
    public abstract void liga();
}

class Moto extends Veiculo {
    public void liga() {

    }
}
```

O método `liga` foi implementado na classe filha, então ela pode ser concreta. Basta pensar que métodos abstratos herdados são *responsabilidades herdadas*: você não poderá ser um objeto concreto enquanto tiver responsabilidades a serem tratadas.

Quando herdamos de uma classe abstrata que possui um método abstrato, temos que escolher: ou implementamos o método, ou somos abstratos também e *passamos adiante* a responsabilidade. Note que a classe pode implementar o método e mesmo assim também ser abstrata.

```
abstract class Veiculo {
    public abstract void liga();
}

abstract class Moto extends Veiculo {

}

class MotoEspecial extends Moto {
    public void liga() {

    }
}

// compila: decidi implementar mas mesmo assim
// manter a classe abstrata
abstract class QuatroRodar extends Veiculo {
    public void liga() {
    }
}
```

Código de uma classe abstrata pode acessar o código da classe concreta, uma vez que ele só será executado quando o objeto for criado:

```
abstract class X{
    abstract void x() {
        System.out.println(y());
    }
    abstract String y();
}
class Y extends X {
    String y() {
        return "codigo";
    }
```

```
}
public class Teste {
    public static void main(String[] args) {
        new X().x(); // imprime código
    }
}
```

Interfaces

Uma interface declara métodos que deverão ser implementados pelas classes concretas que queiram ser consideradas como tal. Por padrão, são todos métodos públicos e abstratos.

```
interface Veiculo {
    void ligar();
    // public abstract! Você pode escrever, mas é por padrão
    // isso.
    public abstract int pegaMarcha();
}
```

Quando você implementa a interface em uma classe concreta, é preciso implementar todos os métodos. Similarmente, ao herdar uma classe abstrata, a classe concreta deve implementar todos os métodos que não foram implementados ainda:

```
// compila, todos os métodos implementados
class Carro implements Veiculo {
    public void ligar() {
    }
    public int pegaMarcha() {
        return 0;
    }
}
// não compila, onde está o pegaMarcha?
class Moto implements Veiculo {
    public void ligar() {
    }
}
// não compila, o método pegaMarcha definiu escopo default,
```

```
// quando deveria definir public
class Triciclo implements Veiculo {
    public void ligar() {
    }
    int pegaMarcha() {
        return 0;
    }
}
```

Valem as mesmas regras de quando você herda de uma classe abstrata: ou você tem todos os métodos reescritos, e aí pode declará-la como concreta, ou então você precisa declará-la como abstrata.

```
// compila, pois a classe é abstrata
abstract class Moto implements Veiculo {
    public void ligar() {
    }
}
```

Uma classe pode implementar diversas interfaces:

```
abstract class MinhaClasse implements Serializable, Runnable {
}
```

Justamente por isso, a prova vê como um bom uso de interfaces quando você quer herdar de dois lugares mas a herança de classes não permite. Para a prova, essa razão é suficiente, mas na prática existe uma diferença grande entre composição (herança de interfaces não envolve herdar comportamento e variáveis membro) e herdar comportamento e variáveis membro de uma classe mãe. Como a implementação de uma interface nos obriga a escrever todos os métodos, estamos compondo nossa classe de diversas interfaces.

Lembre-se que uma interface pode herdar de outra, inclusive de diversas interfaces:

```
interface A extends Runnable {}
interface B extends Serializable {}
interface C extends Runnable, Serializable {}
```

Note que uma interface nunca implementa outra interface:

```
interface A implements Runnable {} // não compila
```

Você pode declarar variáveis em uma interface, todas elas serão `public final static`, isto é, constantes.

```
interface X {
    int i = 5;
    // você até pode escrever public static final, mas é sempre
    // assim
}
```

Uma interface, por sua vez, pode estender outra interface, herdando suas responsabilidades e constantes. Uma interface pode estender mais de uma interface!

```
interface X extends Runnable, Comparable { }
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  abstract class B {
2      void x() {
3          System.out.println(y());
4      }
5      abstract int y();
6  }
7  abstract class C extends B {
8      int y() { return 1; }
9  }
10 class D extends C {
11     int y() { return 2; }
12 }
13 class A {
14     public static void main(String[] args) {
15         D d  = (D) (C) new D();
16         d.x();
17     }
18 }
```

a) Não compila.

b) Compila e imprime ‘1‘.

c) Compila e imprime ‘2‘.

d) Compila e roda com exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    abstract void x() {
        System.out.println(y());
    }
    abstract int y();
}
abstract class C extends B {
    int y() { return 1; }
}
class D extends C {
    int y() { return 2; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime ‘1‘.

c) Compila e imprime ‘2‘.

d) Compila e roda com exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
```

```
5      abstract int y();
6  }
7  abstract class C extends B {
8      abstract int y();
9  }
10 class D extends C {
11     int y() { return 1; }
12 }
13 class A {
14     public static void main(String[] args) {
15         D d  = (D) (C) new D();
16         d.x();
17     }
18 }
```

a) Não compila.

b) Compila e imprime '1'.

c) Compila e imprime '2'.

d) Compila e roda com exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  abstract class B {
2      void x() {
3          System.out.println(y());
4      }
5      int y() {
6          return 2;
7      }
8  }
9  abstract class C extends B {
10     abstract int y();
11 }
12 class D extends C {
13     int y() { return 1; }
14 }
15 class A {
16     public static void main(String[] args) {
```

```
17          D d  = (D) (C) new D();
18          d.x();
19      }
20  }
```

a) Não compila.

b) Compila e imprime '1'.

c) Compila e imprime '2'.

d) Compila e roda com exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1  abstract class B {
2      void x() {
3          System.out.println(y());
4      }
5      final int y() {
6          return 2;
7      }
8  }
9  abstract class C extends B {
10     int y() {
11         return 3;
12     }
13 }
14 class D extends C {
15     int y() { return 1; }
16 }
17 class A {
18     public static void main(String[] args) {
19         D d  = (D) (C) new D();
20         d.x();
21     }
22 }
```

a) Não compila.

b) Compila e imprime '1'.

c) Compila e imprime '2'.

d) Compila e roda com exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    Object y() { return "a"; }
}
abstract class C extends B {
    abstract String y();
}
class D extends C {
    String y() { return "b"; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime 'a'.

c) Compila e imprime 'b'.

d) Compila e roda com exception.

Capítulo 10

Lidando com exceções

10.1 Diferencie entre exceções do tipo checked, runtime e erros

Durante a execução de uma aplicação, erros podem acontecer. A linguagem Java oferece um mecanismo para que o programador possa definir as providências apropriadas a serem tomadas na hora em que um *erro de execução* ocorrer.

Os erros de execução são classificados em algumas categorias. É fundamental que você seja capaz de, dado um erro de execução, determinar seu tipo. A classificação das categorias depende exclusivamente da hierarquia das classes que modelam os erros de execução.

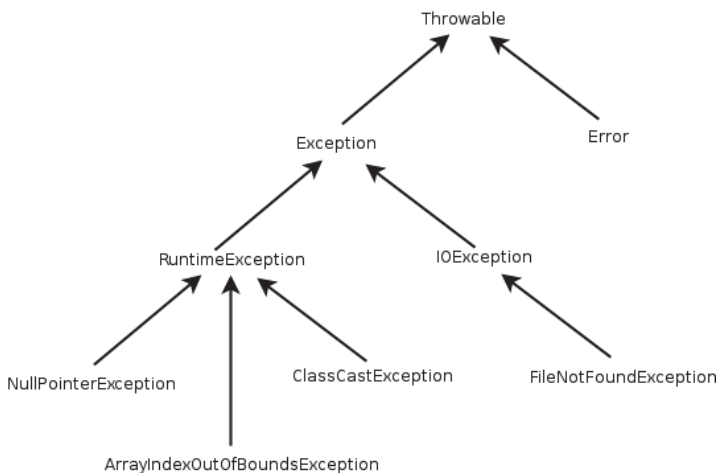
A classe principal dessa hierarquia é a `Throwable`. Qualquer erro de execução é um objeto dessa classe ou de uma que deriva dela.

Como filhas diretas de `Throwable` temos: `Error` e `Exception`.

Os `Errors` são erros de execução gerados por uma situação totalmente anormal que não deveria ser prevista pela aplicação. Por exemplo, um `OutOfMemoryError` é gerado quando a JVM não tem mais memória RAM disponível para oferecer para as aplicações. Em geral, esse tipo de erro não é responsabilidade das aplicações pois quem cuida do gerenciamento de memória é a JVM.

Por outro lado, as `Exceptions` são erros de execução que são de responsabilidade das aplicações, ou seja, são as aplicações que devem tratar ou evitar esses erros. Por exemplo, um `SQLException` é gerado quando algum erro ocorre na comunicação entre a aplicação e o banco de dados. Esse tipo de erro deve ser tratado ou evitado pela aplicação.

Por sua vez, as `Exceptions` são divididas em duas categorias: as **unchecked**s e as **checked**s. As *uncheckeds* são exceptions que teoricamente podem ser mais facilmente evitadas pelo próprio programador se ele codificar de maneira mais cuidadosa. As *checkeds* são exceptions que teoricamente não são fáceis de evitar, de modo que a melhor abordagem é estar sempre preparado para seu acontecimento.

As uncheckeds são definidas pelas classes que derivam de `RuntimeException`, que por sua vez é filha direta de `Exception`. As outras classes na árvore da `Exception` definem as checkeds.

Essas diferenças nao ficam apenas na teoria. O compilador irá verificar se seu programa pode lançar alguma checked exception e, neste caso, obrigá-lo a tratar essa exception de alguma maneira. No caso das exceptions unchecked, não há nenhuma verificação por parte do compilador pelo tratamento ou não.

1) Dentre as classes a seguir qual delas não é checked?

- `java.io.IOException`
- `java.sql.SQLException`
- `java.lang.Exception`
- `java.lang.IndexOutOfBoundsException`
- `java.io.FileNotFoundException`

10.2 Descreva o que são exceções e para que são utilizadas em Java

Imagine a situação em que tentamos acessar uma posição em um array:

```
public void fazAlgo(int[] idades) {
    System.out.println(idades[0]);
}
```

O que acontece se o array enviado para o método é vazio? Se esse código imprimisse nulo ou um número padrão, nesse caso, teríamos sempre que nos preocupar, como em:

```
public void fazAlgo(int[] idades) {
    if(idades[0]==null) return;
    // return para caso o Java devolva nulo ao acessar
    // uma posição inválida

    System.out.println(idades[0]);
}
```

Pense como seria difícil tratar todas as situações possíveis que *fogem do padrão* de comportamento que estamos desejando. Nesse caso, o comportamento padrão, aquilo que acontece 99% das vezes e que esperamos que aconteça é que a posição acessada dentro do array seja válido. Não queremos ter que verificar toda vez se o valor é válido, e não queremos entupir nosso código com diversos `ifs` para diversas condições. As exceções à regra, as *exceptions*, são a alternativa para o controle de fluxo: em vez de usarmos `ifs` para controlar o fluxo que foge do padrão, é possível usar as *exceptions* para esse papel. Veremos adiante como tratar erros, como o acesso a posições inválidas, tentar acessar variáveis com valores inválidos etc.

Caso uma exception estoure e sua *stack trace* seja impressa, teremos algo como:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at SuaClasse.fazAlgo(SuaClasse.java:20)
    at SuaClasse.main(SuaClasse.java:30)
```

Note como a *stack trace* indica que método estava sendo invocado, em qual linha do arquivo fonte está essa invocação, quem invocou este método etc.

O importante é lembrar que as *exceptions* permitem que isolemos o tratamento de um comportamento por blocos, separando o bloco de lógica de nosso negócio do bloco de tratamentos de erros (sejam eles *Exceptions* ou *Errors*, como veremos adiante). O stack trace de uma `Exception` também ajuda a encontrar onde exatamente o problema ocorreu e o que estava sendo executado naquela *Thread* naquele instante.

1) Escolha 2 opções.

   Exception é um mecanismo para...

   - tratar entrada de dados do usuário.
   - que você pode usar para determinar o que fazer quando algo inesperado acontece.
   - que a VM usa para fechar o programa caso algo inesperado aconteça.

- controlar o fluxo da aplicação.
- separar o tratamento de erros da lógica principal.

2) De que maneira a API de exceptions pode ajudar a melhorar o código de seu programa?

   Escolha 2 opções:

   - Permitindo separar o tratamento de erro da lógica do programa.
   - Permitindo tratar o erro no mesmo ponto onde ele ocorre.
   - Permitindo estender as classes que já existem e criar novas exceptions.
   - Disponibilizando várias classes com todas as exceptions possíveis prontas.
   - Aumentando a segurança da aplicação disponibilizando os erros nos logs.

10.3 Crie um bloco try-catch e determine como exceções alteram o fluxo normal de um programa

O programador pode definir um tratamento para qualquer tipo de erro de execução. Antes de definir o tratamento, propriamente, é necessário determinar o trecho de código que pode gerar um erro na execução. Isso tudo é feito com o comando `try-catch`.

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Throwable t) { // pegando todos os possíveis erros de
                        //execução.
    // tratamento para o possível erro de execução.
}
```

A sintaxe do `try-catch` tem um bloco para o programador definir o trecho de código que *pode* gerar um erro de execução. Esse bloco é determinado pela palavra `try`. O programador também pode definir quais tipos de erro ele quer pegar para tratar. Isso é determinado pelo argumento do `catch`. Por fim, o tratamento é definido pelo bloco que é colocado após o argumento do `catch`.

Durante a execução, se um erro acontecer, a JVM redireciona o fluxo de execução da linha do bloco do `try` que gerou o erro para o bloco do `catch`. Importante! As linhas do bloco do `try` abaixo daquela que gerou o erro não serão executadas.

Fazer um `catch` em `Throwable` não é uma boa prática, pois todos os erros possíveis são tratados pela aplicação. Porém, os `Error`s não deveriam ser tratados pela aplicação, já que são de responsabilidade da JVM. Assim, também não é boa prática dar `catch` em `Error`s.

Modificando o argumento do `catch`, o programador define quais erros devem ser pegos para serem tratados.

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Exception e) { // pegando todas as exceptions.
    // tratamento para o possível erro de execução.
}
```

Para a prova, é fundamental saber quando o programador pode ou não pode usar o `try-catch`. A única restrição de uso do `try-catch` envolve as checked exceptions. Qual é a regra? O programador só pode usar `try-catch` em uma checked exception se o código do bloco do `try` pode realmente lançar a checked exception em questão.

```
try {
    System.out.println("não acontece SQLException");
} catch(SQLException e){ // pegando SQLException.
    // tratamento.
}
```

Esse código **não** compila pois o trecho envolvido no bloco do `try` nunca geraria a checked `SQLException`. O compilador avisa com um erro de

"unreachable code". Já o exemplo a seguir compila, pois pode ocorrer um `FileNotFoundException`:

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.FileNotFoundException e){
    // tratamento.
}
```

O código a seguir não tem nenhum problema, pois o programador pode usar o `try-catch` em qualquer situação para os erros de execução que não são checked exceptions.

```
try {
    System.out.println("Ok");
} catch (RuntimeException e) { // pegando RuntimeException
                               // (unckecked).
    // tratamento.
}
```

Podemos pegar tudo, exceptions e erros:

```
try {
    System.out.println("Ok");
} catch (Throwable e) {
    // tratamento
}
```

Quando a exception é pega, o fluxo do programa é sair do bloco `try` e entrar no bloco `catch`, portanto, o código a seguir imprime `peguei` e `continuando normal`:

```
String nome = null;
try {
    nome.toLowerCase();
    System.out.println("segunda linha do try");
} catch(NullPointerException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

Mas, se a exception que ocorre não é a que foi definida no catch, a chamada do método para e volta, jogando a exception como se não houvesse um `try/catch`. O cenário a seguir demonstra essa situação e não imprime nada:

```
String nome = null;
try {
    nome.toLowerCase();
    System.out.println("segunda linha do try");
} catch(IndexOutOfBoundsException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

Lembre-se sempre do polimorfismo, portanto, pegar `IOException` é o mesmo que pegar todas as filhas de `IOException` também. O código a seguir trata o caso de o arquivo não existir além de todas as outras filhas de `IOException`:

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.IOException e){
    // tratamento.
}
```

Bloco finally

Tem coisas que não podemos deixar de fazer em hipótese alguma. Seja no sucesso ou no fracasso, temos obrigação de cumprir com algumas tarefas.

Imagine um método que conecta com um banco de dados. Não importa o que aconteça, no fim desse método a conexão deveria ser fechada. Durante a comunicação com o banco de dados, há o risco de ocorrer uma `SQLException`.

```
void metodo(){
    try {
        abreConexao();
        fazConsultas();
        fechaConexao();
```

```
    } catch (SQLException e) {
        // tratamento
    }
}
```

Nesse código, há um grande problema: se um `SQLException` ocorrer durante as consultas, a conexão com o banco de dados não será fechada. Para tentar resolver esse problema, o bloco do `catch` poderia invocar o método `fechaConexao()`. Então, se acontecesse um `SQLException` o bloco do `catch` seria executado e, consequentemente, a conexão seria fechada.

Mas ainda não solucionamos o problema, pois outro tipo de erro poderia acontecer nas consultas. Por exemplo, uma `NullPointerException` que não está sendo tratada. Para resolver o problema de fechar a conexão, um outro recurso do Java será utilizado, o bloco **finally**. Esse bloco é sempre executado, tanto no sucesso quanto no fracasso por qualquer tipo de erro.

```
void metodo(){
    try {
        abreConexao();
        fazConsultas(); // Não precisa mais fechar a conexao
                        // aqui.
    } catch(SQLException e) {
        // tratamento
    } finally {
        fechaConexao(); // fechando a conexao no sucesso ou no
                        // fracasso.
    }
}
```

Para melhor entender o fluxo do `try-cacth` com o `finally`, veja o próximo exemplo.

```
class A {
    void metodo() {
        try{
            //A
            //B
        }catch(SQLException e){
```

```
            //C
        }finally{
            //D
        }
        //E
    }
}
```

- Em uma execução normal, sem erros nem exceções, ele executaria `A`, `B`, `D`, `E`.
- Com `SQLException` em `A`, ele executaria `C`, `D`, `E`.
- Com `NullPointerException` em `A`, ele executaria apenas `D` e sairia.
- Se `A` fosse um `System.exit(0);`, ele apenas executa `A` e encerra o programa.
- Se ocorresse um erro `A`, executaria apenas `D` (dependendo do erro).

Uma outra maneira um pouco menos convencional de usar o `finally` é sem o bloco `catch`, como no exemplo a seguir.

```
class A{
  void metodo() {
    try {
      System.out.println("imprime algo");
    } finally {
      // sempre permite fechar
    }
  }
}
```

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String nome;
```

```
4         try {
5             nome.toLowerCase();
6             System.out.println("a");
7         } catch(NullPointerException ex) {
8             System.out.println("b");
9         }
10        System.out.println("c");
11    }
12 }
```

a) Não compila.

b) Compila e, ao rodar, imprime "abc".

c) Compila e, ao rodar, imprime "bc".

d) Compila e, ao rodar, imprime "a".

e) Compila e, ao rodar, imprime "b".

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String nome = null;
4         try {
5             nome.toLowerCase();
6             System.out.println("a");
7         } catch(NullPointerException ex) {
8             System.out.println("b");
9         }
10        System.out.println("c");
11    }
12 }
```

a) Não compila..

b) Compila e, ao rodar, imprime "abc".

c) Compila e, ao rodar, imprime "bc".

d) Compila e, ao rodar, imprime "a".

e) Compila e, ao rodar, imprime "b".

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         String nome;
4         try {
5             nome.toLowerCase();
6             System.out.println("a");
7         } catch(NullPointerException ex) {
8             System.out.println("b");
9         } finally {
10            System.out.println("c");
11        }
12        System.out.println("d");
13    }
14 }
```

a) Não compila.

b) Compila e, ao rodar, imprime "abcd".

c) Compila e, ao rodar, imprime "bcd".

d) Compila e, ao rodar, imprime "ac".

e) Compila e, ao rodar, imprime "bc".

f) Compila e, ao rodar, imprime "ad".

g) Compila e, ao rodar, imprime "bd".

10.4 Invoque um método que joga uma exceção

Eventualmente, um método qualquer não tem condição de tratar um determinado erro de execução. Nesse caso, esse método pode deixar passar o erro para o próximo método na pilha de execução.

Para deixar passar qualquer erro de execução que não seja uma checked exception, é muito simples: basta não fazer nada.

```
class Teste {
```

```
    void primeiro(){
        System.out.println("primeiro antes");
        this.segundo();
        System.out.println("primeiro depois");
    }

    void segundo() {
        String s = null;
        System.out.println("segundo antes");
        s.length();
        System.out.println("segundo depois");
    }
}
```

O segundo método declara uma variável não primitiva e a inicializa com `null`. Logo em seguida, ele utiliza o operador `.` em uma referência que sabemos estar nula. Nesse ponto, na hora da execução, um `NullPointerException` é gerado. Perceba que não há `try-catch` no segundo método, então ele não está pegando e tratando o erro, mas sim deixando-o passar. O primeiro método não define o `try-catch`, ou seja, também deixa passar o `NullPointerException`. O resultado é a impressão de `primeiro antes`, `segundo antes`.

Agora, para deixar passar uma checked exception, o método é obrigado a deixar explícito (avisado) que pretende deixar passar. Na assinatura do método, o programador pode deixar avisado que pretende deixar passar determinados erros de execução. Isso é feito através da palavra-chave `throws`.

```
class Teste {

    void primeiro(){
        try {
            System.out.println("primeiro antes");
            this.segundo();
            System.out.println("primeiro depois");
        } catch(IOException e) {
            // tratamento.
            System.out.println("primeiro catch");
        }
```

```
        System.out.println("primeiro fim");
    }

    void segundo() throws IOException {
        System.out.println("segundo antes");
        System.in.read(); // pode lançar IOException
        System.out.println("segundo depois");
    }
}
```

O segundo método invoca o `read()` no `System.in`. Essa invocação pode gerar um `IOException`, de modo que o segundo método tem duas alternativas: ou pega e trata o possível erro ou o deixa passar. Para deixar passar, o comando `throws` deve ser utilizado na sua assinatura do segundo método. Isso indicará que um `IOException` pode ser lançado.

Dessa forma, o primeiro método que invoca o segundo pode receber uma `IOException`. Então, ele também tem duas escolhas: ou pega e trata usando `try-catch`, ou deixa passar usando o `throws`. O resultado é a impressão de `primeiro antes`, `segundo antes`, `primeiro catch` e `primeiro fim`.

Gerando um erro de execução

Qualquer método, ao identificar uma situação errada, pode criar um erro de execução e lançar para quem o chamou. Vale lembrar que os erros de execução são representados por objetos criados a partir de alguma classe da hierarquia da classe `Throwable`, logo, basta o método instanciar um objeto de qualquer uma dessas classes e depois lançá-lo.

Se o erro não for uma checked exception, basta criar o objeto e utilizar o comando `throw` para lançá-lo na pilha de execução (não confunda com o `throws`):

```
class Teste {

    void primeiro(){
        try {
            this.segundo();
```

```
        } catch (RuntimeException e) {
            // tratamento.
        }
    }

    void segundo() {
        throw new RuntimeException();
    }
}
```

Se o erro for uma checked exception, é necessário também declarar na assinatura do método o comando `throws`:

```
class Teste {

    void primeiro(){
        try {
            this.segundo();
        } catch(Exception e) {
            // tratamento.
        }
    }

    void segundo() throws Exception {
        throw new Exception();
    }
}
```

Podemos ainda criar nossas próprias exceções, bastando criar uma classe que entre na hierarquia de `Throwable`.

```
class MinhaException extends Exception{}
```

Em qualquer lugar do código, é opcional o uso do `try` e `catch` de uma unchecked exception para compilar o código. Em uma checked exception, é obrigatório o uso do `try/catch` ou `throws`.

O exemplo a seguir mostra uma unchecked exception sendo ignorada e o erro vazando, e nada será impresso:

```
public class Teste {

    public static void main(String[] args) {
        metodo();
        System.out.println("Apos a invocacao do metodo");
    }

    private static void metodo() {
        int[] i= new int[10];
        System.out.println(i[15]);
        System.out.println("Apos a exception");
    }

}
```

Ao pegarmos a exception, será impresso também "Apos a invocacao do metodo" uma vez que após o catch, o fluxo volta ao normall:

```
public class Teste {

    public static void main(String[] args) {
        try {
            metodo();
        } catch(RuntimeException ex) {
            System.out.println("Exception pega");
        }
        System.out.println("Apos a invocacao do metodo");
    }

    private static void metodo() {
        int[] i= new int[10];
        System.out.println(i[15]);
        System.out.println("Apos a exception");
    }

}
```

Podemos ter também múltiplas expressões do tipo `catch`. Nesse caso, será invocada somente a cláusula adequada, e não as outras. No código a

seguir, se o `metodo2` jogar uma `ArrayIndexOutOfBoundsException`, será impresso `runtime`:

```
void metodo1() {
    try {
        metodo2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(RuntimeException ex) {
        System.out.println("runtime");
    } catch(Exception ex) {
        System.out.println("exception qualquer");
    }
}
```

E a ordem faz diferença? Sim, o Java procura o primeiro `catch` que pode trabalhar a `Exception` adequada.

Repare que `RuntimeException` herda de `Exception` e, portanto, deve vir antes da mesma na ordem de *catches*.

Caso ela viesse depois, ela nunca seria invocada, pois o Java verificaria que toda `RuntimeException` é `Exception` e `Exception` teria tratamento de preferência (por sua ordem). O exemplo a seguir não compila por este motivo:

```
void metodo1() {
    try {
        metodo2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(Exception ex) {
        System.out.println("exception qualquer");
    } catch(RuntimeException ex) {
        // não compila pois jamais será executado
        System.out.println("runtime");
    }
}
```

Cuidado também com exceptions nos inicializadores:

```
class AcessoAoArquivo {
    // não compila, pois ao instanciar, pode dar IOException,
```

```
    // mas o construtor não fala nada
    private InputStream is = new FileInputStream("entrada.txt");
}
```

Nesses casos, precisamos dizer no construtor que a `Exception` pode ser jogada:

```
class AcessoAoArquivo {
    private InputStream is = new FileInputStream("entrada.txt");

    AcessoAoArquivo() throws IOException{
        // estou avisando os clientes dessa classe
        // que ao instanciar pode dar essa exception
        // e agora compila
    }
}
```

1) Qual classe podemos colocar no código a seguir para que ele compile?

```
1 import java.io.*;
2 class X {
3     InputStream y() throws NOME_AQUI {
4         return new FileInputStream("a.txt");
5     }
6     void z() throws NOME_AQUI{
7         InputStream is = y();
8         is.close();
9     }
10 }
```

* `java.io.IOException` * `java.sql.SQLException` * `java.lang.Exception` * `java.lang.IndexOutOfBoundsException` * `java.io.FileNotFoundException`

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     void m2() {
3         System.out.println("e");
```

```
4         int[][]x = new int[15][20];
5         System.out.println("f");
6     }
7     void m() {
8         System.out.println("c");
9         m2();
10        System.out.println("d");
11    }
12    public static void main(String[] args) {
13        System.out.println("a");
14        new A().m();
15        System.out.println("b");
16    }
17 }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
1 class A {
2     void m2() {
3         System.out.println("e");
4         int[] x = new int[15];
5         x[20] = 13;
6         System.out.println("f");
7     }
8     void m() {
9         System.out.println("c");
10        m2();
11        System.out.println("d");
12    }
13    public static void main(String[] args) {
14        System.out.println("a");
```

```
15          new A().m();
16          System.out.println("b");
17      }
18  }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1   class A {
2       void m2() {
3           System.out.println("e");
4           new java.io.FileInputStream("a.txt");
5           System.out.println("f");
6       }
7       void m() {
8           System.out.println("c");
9           m2();
10          System.out.println("d");
11      }
12      public static void main(String[] args) {
13          System.out.println("a");
14          new A().m();
15          System.out.println("b");
16      }
17  }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1 class A {
2     void m2() throws java.io.FileNotFoundException {
3         System.out.println("e");
4         new java.io.FileInputStream("a.txt");
5         System.out.println("f");
6     }
7     void m() throws java.io.IOException {
8         System.out.println("c");
9         m2();
10        System.out.println("d");
11    }
12    public static void main(String[] args)
13                throws java.io.FileNotFoundException {
14        System.out.println("a");
15        new A().m();
16        System.out.println("b");
17    }
18 }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1 class A {
2     void m2() throws java.io.FileNotFoundException {
3         System.out.println("e");
4         new java.io.FileInputStream("a.txt");
```

```
5          System.out.println("f");
6      }
7      void m() throws java.io.FileNotFoundException {
8          System.out.println("c");
9          m2();
10         System.out.println("d");
11     }
12     public static void main(String[] args) throws
13     java.io.IOException {
14         System.out.println("a");
15         new A().m();
16         System.out.println("b");
17     }
18 }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1  class A {
2      void m2() throws java.io.FileNotFoundException {
3          System.out.println("e");
4          new java.io.FileInputStream("a.txt");
5          System.out.println("f");
6      }
7      void m() throws java.io.FileNotFoundException {
8          System.out.println("c");
9          try {
10             m2();
11         } catch(java.io.FileNotFoundException ex) {
12         }
13         System.out.println("d");
```

```
14      }
15      public static void main(String[] args) throws
16      java.io.IOException {
17          System.out.println("a");
18          new A().m();
19          System.out.println("b");
20      }
21  }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `acedb`.

f) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1   class MyException extends RuntimeException {
2
3   }
4   class A {
5       void m2() throws java.io.FileNotFoundException {
6           System.out.println("e");
7           new MyException();
8           System.out.println("f");
9       }
10      void m() throws java.io.FileNotFoundException {
11          System.out.println("c");
12          try {
13              m2();
14          } catch(java.io.FileNotFoundException ex) {
15          }
16          System.out.println("d");
17      }
18      public static void main(String[] args) throws
```

```
19      java.io.IOException {
20          System.out.println("a");
21          new A().m();
22          System.out.println("b");
23      }
24  }
```

a) Não compila.

b) Compila e imprime acefdb.

c) Compila e imprime ace e joga uma Exception.

d) Compila e imprime acedb e joga uma Exception.

e) Compila e imprime ace, joga uma Exception e imprime db.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1  class MyException extends RuntimeException {
2
3  }
4  class A {
5      void m2() throws java.io.FileNotFoundException {
6          System.out.println("e");
7          throws new MyException();
8          System.out.println("f");
9      }
10     void m() throws java.io.FileNotFoundException {
11         System.out.println("c");
12         try {
13             m2();
14         } catch(java.io.FileNotFoundException ex) {
15         }
16         System.out.println("d");
17     }
18     public static void main(String[] args) throws
19     java.io.IOException {
20         System.out.println("a");
21         new A().m();
```

```
22          System.out.println("b");
23      }
24  }
```

a) Não compila.
b) Compila e imprime `acefdb`.
c) Compila e imprime `ace` e joga uma `Exception`.
d) Compila e imprime `acedb` e joga uma `Exception`.
e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
1  class MyException extends RuntimeException {
2
3  }
4  class A {
5      void m2() throws java.io.FileNotFoundException {
6          System.out.println("e");
7          boolean sim = true;
8          if(sim) throws new MyException();
9          System.out.println("f");
10     }
11     void m() throws java.io.FileNotFoundException {
12         System.out.println("c");
13         try {
14             m2();
15         } catch(java.io.FileNotFoundException ex) {
16         }
17         System.out.println("d");
18     }
19     public static void main(String[] args) throws
20     java.io.IOException {
21         System.out.println("a");
22         new A().m();
23         System.out.println("b");
24     }
25 }
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

11) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class MyException extends RuntimeException {

}
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        boolean sim = true;
        if(sim) throw new MyException();
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

10.5 Reconheça classes de exceções comuns e suas categorias

Para a prova, é necessário conhecer algumas exceptions clássicas do Java. Na sequência, vamos conhecer essas exceptions e entender em que situações elas ocorrem.

ArrayIndexOutOfBoundsException e IndexOutOfBoundsException

Um `ArrayIndexOutOfBoundsException` ocorre quando se tenta acessar uma posição que não existe em um array.

```
class Teste {
    public static void main(String[] args) {
        int[] array = new int[10];
        array[10] = 10; // Aqui ocorre
                        // ArrayIndexOutOfBoundsException.
    }
}
```

Da mesma maneira, quando tentamos acessar uma posição não existente em uma lista , a exception é diferente, no caso `IndexOutOfBoundsException`:

```
class Teste {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<String>();

        //Aqui ocorre IndexOutOfBoundsException
        String valor = lista.get(2);
    }
}
```

NullPointerException

Toda vez que o operador . é utilizado em uma referência nula, um NullPointerException é lançado.

```
class Teste {
    public static void main(String[] args) {
        String s = null;
        s.length(); // Aqui ocorre uma NullPointerException
    }
}
```

ClassCastException

Quando é feito um casting em uma referência para um tipo incompatível com o objeto que está na memória em tempo de execução, ocorre um ClassCastException.

```
class Teste {
    public static void main(String[] args) {
        Object o = "SCJP"; // String
        Integer i = (Integer)o; // Aqui ocorre
                                // ClassCastException.
    }
}
```

NumberFormatException

Um problema comum que o programador enfrenta no dia a dia é ter que "transformar" texto em números. A API do Java oferece diversos métodos para tal tarefa. Porém, em alguns casos não é possível "parsear" o texto, pois ele pode conter caracteres incorretos.

```
class Teste {
    public static void main(String[] args) {
        String s = "ABCD1";

        // Aqui ocorre um NumberFormatException.
        int i =    Integer.parseInt(s);
```

```
    }
}
```

IllegalArgumentException

Qualquer método deve verificar se os valores passados nos seus parâmetros são válidos. Se um método constata que os parâmetros estão inválidos, ele deve informar quem o invocou que há problemas nos valores passados na invocação. Para isso, é aconselhado que o método lance `IllegalArgumentException`.

```
class Teste {
    public static void main(String[] args) {
        try {
            divideEImprime(5,0);
        } catch (IllegalArgumentException e) {
            // tratamento.
        }
    }

    public static void divideEImprime(int i, int j) {
        if(j == 0) { // Evita dividir por zero.
            throw new IllegalArgumentException();
        }
        System.out.println(i/j);
    }
}
```

IllegalStateException

Suponha que uma pessoa possa fazer três coisas: dormir, acordar e andar. Para andar, a pessoa precisa estar acordada. A classe `Pessoa` modela o comportamento de uma pessoa. Ela contém um atributo `boolean` que indica se a pessoa está acordada ou dormindo e um método para cada coisa que uma pessoa faz (`dormir()`, `acordar()` e `andar()`).

O método `andar()` não pode ser invocado enquanto a pessoa está dormindo. Mas, se for, ele deve lançar um erro de execução. A biblioteca do Java já tem uma classe pronta para essa situação, a classe é a

`IllegalStateException`. Ela significa que o estado atual do objeto não permite que o método seja executado.

```
class Pessoa {
    boolean dormindo = false;

    void dormir() {
        this.dormindo = true;
        System.out.println("dormindo...");
    }
    void acordar() {
        this.dormindo = false;
        System.out.println("acordando...");
    }
    void andar() {
        if(this.dormindo) { // Só pode andar acordado.
            throw new IllegalStateException("Deveria estar
                                             acordado!");
        }

        System.out.println("andando...");
    }
}
```

ExceptionInInitializerError

No momento em que a máquina virtual é disparada, ela não carrega todo o conteúdo do *classpath*, em outras palavras, ela não carrega em memória todas as classes referenciadas pela sua aplicação.

Uma classe é carregada no momento da sua primeira utilização. Isso se dá quando algum método estático ou atributo estático são acessados ou quando um objeto é criado a partir da classe em questão.

No carregamento de uma classe, a JVM pode executar um trecho de código definido pelo programador. Esse trecho deve ficar no que é chamado bloco estático.

```
class A {
    static {
```

```
        // trecho a ser executado no carregamento da classe.
    }
}
```

É totalmente possível que algum erro de execução seja gerado no bloco estático. Se isso acontecer, a JVM vai "embrulhar" esse erro em um `ExceptionInInitializerError` e dispará-lo.

Esse erro pode ser gerado também na inicialização de um atributo estático se algum problema ocorrer. Exemplo:

```
class A {
    static {
        if(true)
            throw new RuntimeException("nao vou deixar nao...");
    }
}

public class Teste {

    public static void main(String[] args) {
        new A();
    }
}
```

Gera o erro de inicialização:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
    at Teste.main(Teste.java:11)
Caused by: java.lang.RuntimeException: nao vou deixar nao...
    at A.<clinit>(Teste.java:4)
    ... 1 more
```

```
class P{
    static int a = Integer.parseInt("a");
}
```

StackOverflowError

Todos os métodos invocados pelo programa Java são empilhados na **Pilha de Execução**. Essa pilha tem um limite, ou seja, ela pode estourar:

```
class Teste {
    public static void main(String[] args) {
        metodoSemFim();
    }

    static void metodoSemFim() {
        metodoSemFim();
    }
}
```

Repare que, nesse exemplo, o `metodoSemFim()` chama ele mesmo (recursão). Do jeito que está, os métodos serão empilhados eternamente e a pilha de execução vai estourar.

NoClassDefFoundError

Na etapa de compilação, todas as classes referenciadas no código-fonte precisam estar no *classpath*. Na etapa de execução também. O que será que acontece se uma classe está no *classpath* na compilação mas não está na execução? Quando isso acontecer será gerado um `NoClassDefFoundError`.

Para gerá-lo, podemos criar um arquivo com duas classes onde uma referencia a outra:

```
class OutraClasse {

}
class Teste {
    public static void main(String[] args) {
        new OutraClasse();
    }
}
```

Compilamos o arquivo, gerando dois arquivos `.class`. Aí apagamos o arquivo `OutraClasse.class`. Pronto, o Java não será capaz de encontrar a classe, dando um erro, `NoClassDefFoundError`.

OutOfMemoryError

Durante a execução de nosso código, o Java vai gerenciando e limpando a memória usada por nosso programa automaticamente, usando o **garbage collector** (GC). O GC vai remover da memória todas as referências de objetos que não são mais utilizados, liberando o espaço para novos objetos. Mas o que acontece quando criamos muito objetos, e não os liberamos? Nesse cenário, o GC não vai conseguir liberar memória, e eventualmente a memória livre irá acabar, ocasionando um `OutOfMemoryError`.

O código para fazer um erro do gênero é simples, basta instanciar infinitos objetos, sem permitir que o garbage collector jogue-os fora. Fazemos isso com `String`s para que o erro aconteça logo:

```
void metodo() {
    ArrayList<String> objetos = new ArrayList<String>();
    String atual = "";
    while(true) {
        atual += " ficou maior";
        objetos.add(atual);
    }
}
```

1) Escolha a opção adequada que indica o `Throwable` que ocorrerá no código a seguir:

```
1 class A {
2     public static void main(String[] args) {
3         main(args);
4     }
5 }
```

a) `IndexOutOfBoundsException`

b) `ArrayIndexOutOfBoundsException`

c) `NullPointerException`

d) `OutOfMemoryError`

e) `StackOverflowError`

f) `ExceptionInInitializationError`

2) Escolha a opção adequada que indica o `Throwable` que ocorrerá no código a seguir:

```
import java.util.*;
class A {
    public static void main(String[] args) {
        ArrayList<String> strings = new ArrayList<String>();
        for(int i=0;i<10;i++)
            for(int j=0;j<10;i++)
                strings.add("string " + i + " " + j);
        System.out.println(strings.get(99999));
    }
}
```

a) `IndexOutOfBoundsException`

b) `ArrayIndexOutOfBoundsException`

c) `NullPointerException`

d) `OutOfMemoryError`

e) `StackOverflowError`

f) `ExceptionInInitializationError`

Capítulo 11

Boa prova

Agora que você acabou todo o livro, está na hora de revisar os pontos em que ficou com dúvida. Refaça os exercícios para reforçar o conteúdo e faça diversos simulados como os citados no começo do livro.

Para efetuar a prova, primeiro compre o voucher no site da Oracle. Busque no site dela pelo código da prova que deseja fazer, a Java SE 7 Programmer I (1Z0-803), adicione a mesma ao carrinho e efetue a compra.

Hoje o pagamento é feito via boleto enviado por email, tome cuidado pois ele pode cair em sua caixa de spam, e tome ainda mais cuidado para não pagar um boleto de spam. Depois de confirmado o recebimento por parte da Oracle você receberá um código de confirmação com o qual será capaz de agendar a data e local de sua prova.

Lembre-se de reler os pontos que tem mais dificuldade no dia que antecede a prova.

Mais uma vez, boa prova. Desejo que todo esse processo tenha aberto sua mente sobre como a linguagem Java funciona, seus limites e suas características gerais.

Depois dessa certificação, que venha a próxima, que exige ainda mais conhecimento da linguagem e de APIs fundamentais da mesma.

Capítulo 12

Respostas dos Exercícios

3.1 - Exercícios

1. A resposta certa é (c). A variável `i` declarada no `for` só é visível dentro do `for`.

2. A resposta certa é (c), a variável `x` declarada como parâmetro do método `main` efetua um shadowing. Nesse instante, ao dizermos `x = 200`, tentamos atribuir um `int` a um array de `String`, erro de compilação.

3. A resposta certa é (d). Isso porque o acesso à variável estática pode ser feito através da instância da classe ou diretamente caso seja uma variável estática sendo acessada por um método estático.

3.2 - Exercícios

1. A resposta certa é (a). O arquivo não compila pois não podemos ter um `import` após a definição de uma classe.

2. A resposta certa é (e). O código compila e roda, não imprimindo nada, pois não chamamos o método `Teste`.

3. A resposta certa é (c). O código compila sem erros: a ordem `package` e `import` está adequada, e os tipos são opcionais dentro de um arquivo java.

4. A resposta certa é (d). Uma vez que o arquivo chama `A.java`, o único tipo público que pode existir dentro dele deve se chamar `A`, o que não é verdade: tentamos definir um tipo `B` público.

3.3 - Exercícios

1. O método `main` não pode devolver `int` nem `Void`. Ele também deve ser público e só receber um argumento: um array (ou varargs) de `String`. Portanto:

   ```
   public static void main(String... args)
   ```

   A resposta certa é (a).

2. Para compilar, estamos trabalhando com arquivos e diretórios, portanto `javac b/A.java`; enquanto, para rodar, estamos pensando em pacotes e classes: `java b.A`. A resposta certa é (d).

3. Ao rodar sem argumentos, nosso array tem tamanho zero, portanto, ao tentar acessar seu primeiro elemento recebemos um `NullPointerException` na linha 5. A resposta certa é (c).

4. Para rodar um programa dentro de um `jar` sem ter um manifesto, devemos usar o classpath customizado. Colocamos o `jar` no classpath e dizemos qual classe desejamos rodar: `java -cp programa.jar b.A`. A resposta certa é (f).

5. Durante a compilação, para adicionar o arquivo `programa.jar` ao classpath, devemos usar `-cp programa.jar` e, para especificar o arquivo adequado, usamos `b/A.java`. A opção que apresenta essas duas características é `javac -cp programa.jar:. b/A.java`. A resposta certa é (h).

3.4 - Exercícios

1. A resposta certa é (a). Ocorre um erro de compilação na classe `Teste` ao tentar importar uma classe não acessível a outros pacotes.

2. A resposta certa é (a). O erro de ambiguidade é dado no `import` e não na utilização, portanto o arquivo não compila.

3. A resposta certa é (c). Não existe ambiguidade uma vez que o import específico tem preferência em cima do `*` .

4. A resposta certa é (c). Não há erro de ambiguidade, simplesmente um import é desnecessário e não gera erro nenhum, apenas um *warning*.

5. A resposta certa é (e). O arquivo B compila pois é uma classe normal. O arquivo C não compila pois tenta acessar B, que está em outro pacote, mas lembre-se que devemos acessar os pacotes diretamente, não existe subpacote. O mesmo vale para A. Portanto, nem A nem C compilam.

6. A resposta certa é (d). Um pacote pode ter nome começando com maiúsculo, isso não afeta em nada. Mas não é o padrão. O nome de variáveis locais e parâmetros não afetam a assinatura de um método em Java. Uma classe não precisa ser pública para ser rodada. Portanto o código compila e roda.

7. A resposta certa é (b). Não importamos a classe A, somente seus membros, erro de compilação ao tentar referenciá-la na linha 5.

8. A resposta certa é (a). `import static` é o uso adequado, e não `static import`, erro na linha 3.

9. A resposta certa é (a). `B` não compila pois tenta acessar uma classe do pacote padrão (sem nome). Classes do pacote padrão só podem ser acessadas por outros tipos do pacote padrão. Não compila.

4.1 - Exercícios

1. A resposta certa é (d). O código compila e imprime 100. Podemos ter espaços em branco desde que não quebre uma palavra-chave, nome de método, classe etc. ao meio. Onde pode ter um espaço em branco, pode haver vários.

2. A resposta certa é (a). O código não compila pois tentamos acessar a variável `idade` que pode não ter sido inicializada. Não é certeza (somente se cair no `if` ela será inicializada).

3. A resposta certa é (f). Não compila, do lado direito da atribuição temos um array de boolean e do lado esquerdo uma variável simples do tipo boolean.

4. A resposta certa é (b). Imprime `false` pois um array de tipos primitivos após a inicialização tem seus valores com o valor padrão do tipo. Para numéricos é 0, para `boolean` é `false` e para referências é `null`.

5. A resposta certa é (b). Não compila pois `boolean` em Java só pode ser `false` ou `true`.

6. A resposta certa é (a). O número octal 09 não existe. Você não precisa aprender a transformar uma base em outra, mas é importante lembrar que binários são compostos de 0s e 1s, octais são compostos de 0s até 7s, hexadecimais são de 0s até 9s e As até Fs (maiúsculo ou minúsculo). O caractere _ é permitido desde que dos dois lados dele tenhamos algarismos válidos, que é o caso de 1_000. Portanto, o único número inválido é 09 (por curiosidade, o número 9 em base octal é 011).

7. A resposta certa é (c). Compila e imprime o alfabeto pois caracteres são números em Java.

8. A resposta certa é (e). `instanceOf` não é palavra reservada: note a letra maiúscula no meio dela. **Nenhuma** palavra-chave em Java possui caractere maiúsculo.

9. A resposta certa é (d). Compila e roda, não imprimindo nada. Lembre-se que os identificadores são `case-sensitive`.

4.2 - Exercícios

1. A resposta certa é (e). Imprime 47, pois a atribuição é por cópia do valor.

2. A resposta certa é (f). Imprime 48, pois a atribuição de objetos é feita por cópia da referência, criamos somente um único objeto do tipo `B`.

4.3 - Exercícios

1. A resposta certa é (d). Não existe conflito de nomes entre variável membro e método ou variável membro e variável local. Ao invocar o método `c`, por causa do `shadowing` da variável `c`, não acessamos a variável membro, sem alterá-la. O resultado é a impressão dos valores 10 e 10 novamente.

4.4 - Exercícios

1. A resposta certa é (c). Compila e não podemos falar nada.

2. A resposta certa é (b). Somente 10 objetos podem ser garbage coletados pois o último continua referenciado pela variável `b`.

3. A resposta certa é (b). O código compila, mas como não chamamos nenhum construtor, o único objeto criado que se assemelha a `B`, porém não é `B`, é um array do tipo `B`, com 100 espaços. Nenhum objeto é criado. Note que para criar devemos, por padrão, invocar o construtor.

4.5 - Exercícios

1. A resposta certa é (b). O código compila e devido à regra de sempre invocar o mais específico, ele sempre invoca o método sem argumentos. Portanto, o resultado é vazio/vazio. Lembre-se que em Java a ordem de definição de métodos não importa para a invocação. Já a ordem das variáveis pode importar, caso uma dependa da outra.

2. A resposta certa é (c). O código compila e imprime 2.

3. A resposta certa é (e). O código compila e imprime 2.

4. A resposta certa é (e). O código compila e imprime 2. Esse é o caso absurdo onde o array é tanto um `Object` quanto um array de `Object`. Por padrão o Java tratará como um array de `Object`.

4.6 - Exercícios

1. A resposta certa é (e). O array começa na posição 0, portanto, o primeiro caractere removido se encontra na posição 2, o `i`. Ele remove todos os caracteres até a posição 3, exceto o da posição 3, portanto somente o `i` é removido.

2. A resposta certa é (a). Os dois métodos retornam `-1` quando não encontram nada, portanto, o segundo resultado é `-2`. Como a posição começa em `0`, o resultado das letras `e` são `5` e `8`, totalizando `13`.

4.7 - Exercícios

1. A resposta certa é (a). Não compila, pois `length()` é um método de `String`, diferente dos arrays em que `length` é um atributo.

2. A resposta certa é (b). Dá `NullPointerException`! msg é `null` e não dá pra chamar `isEmpty` em `null`.

3. A resposta certa é (a). Não compila pois a variável não foi inicializada.

4. A resposta certa é (a). 'Caelum' e 'Caelum - Ensino e Inovação'.
5. A resposta certa é (e). Compila e imprime `Bem-vindo null`
6. A resposta certa é (b). Não compila pois a String não foi inicializada
7. A resposta certa é (b). Não compila por outro motivo: a variável `vazio` não é estática.
8. A resposta certa é (e). Compila e imprime `Bem-vindo null`.
9. A resposta certa é (d). Dá `NullPointerException` ao tentar criar a segunda `String`.
10. A resposta certa é (c). O código compila e imprime `uda`.
11. A resposta certa é (a). Não compila pois `String` possui diversos construtores que recebem um argumento: o compilador não sabe qual deles você deseja invocar pois os tipos que são argumentos do construtor não possuem herança entre si (um não herda necessariamente do outro).
12. A resposta certa é (c). Nenhuma das alternativas dadas com número, pois primeiro ele soma `valor` e `dividePor`, imprimindo 14. A conta de divisão é feita entre `int`, devolvendo um `int` de valor 2. Quando esse número é atribuído a um `double`, continua sendo 2. Portanto, imprime `14` e `2.0`.
13. A resposta certa é (a). O código não compila pois o método `replace` possui duas maneiras de ser invocado: com dois `chars` ou com duas `Strings`. Foram passados uma `String` e um `char`, método que não existe.
14. A resposta certa é (a). Pensamos que pode ser `gualherme`, mas lembramos que `String` em Java é imutável e ela não foi reatribuída. O = dá uma impressão de reatribuição de parte da `String`, mas isso não existe em Java, o lado esquerdo de uma atribuição deve ser sempre uma variável e não uma chamada a um método. Por isso, a linha do `substring` não compila.

5.1 - Exercícios

1. As opções (a) e (c) não compilam e precisam do casting mesmo com `short` e `char` tendo 2 bytes.

   Na opção (a), pode ocorrer de o `short` ser negativo e, portanto, não caber no intervalo dos `chars`. Na opção (c), o `char` pode ser muito grande e sair fora do alcance dos positivos do `short`.

   A opção (b) compila, pois o `char` que possui 2 bytes pode ser atribuído para um `long` que possui 8 bytes.

2. A resposta certa é (b). Análise linha a linha:

   – divisão inteira: i1 vale 1
   – divisão inteira, que depois é promovido a `double`: i2 vale 1.0
   – divisão `double`: i3 vale 1.5
   – x vale 0L e d vale 0.0 (duas promoções)
   – O resultado é 3.5

3. A resposta certa é (d). Mesmo `c` sendo `null`, por estarmos usando o operador `&`, a segunda parte da expressão (`c.getPreco() > 10000`) será avaliada, causando uma `NullPointerException` na chamada do método `getPreco()` caso `c` seja `null`. Poderíamos evitar isso usando o operador de curto-circuito, `&&`.

4. A resposta certa é (d): i e c.

5. A resposta certa é (d). Pode-se utilizar o operador booleano de ou exclusivo `^`:

```
if (trem ^ carro) {
    // ....
}
```

6. 1ª linha: `ArithmeticException: / by zero` 2ª linha: `Infinity` 3ª linha: `Infinity` 4ª linha: `-Infinity`

7. A resposta certa é (b). O código não compila na linha 7. O compilador não tem certeza se a variável `y` vai ser iniciada sempre. Como a declaração é feita e o único valor atribuído é dentro do `for`, o compilador não tem certeza se o `for` vai ser executado mesmo.

8. A resposta certa é (d).

9. A resposta certa é (e). Compila, roda e imprime A75. Cuidado ao compilar e rodar pois alguns caracteres podem precisar ser escapados pelo seu `shell` ou `bash` (não cobrado na prova).

5.1 - Exercícios

1. Não compila! Toda conta devolve no mínimo um `int`. O resultado de `b1 + b2` é `int`. Podemos fazer casting ou declarar `b3` como `int`.

5.1 - Exercícios

1. A resposta certa é (c). Não compila, pois toda conta devolve no mínimo um `int`, e um `int` não cabe em um `byte`.

5.1 - Exercícios

1. A resposta certa é (c), 3.

   O `for` externo vai contar de 0 a 5, mas dentro do `for` tem um `if` que pré-incrementa o `i`. Esse `if` vai quebrar o loop no momento que o valor retornado for divisível por 3, isto é, quando `i` valer 3 nesse caso.

2. A resposta certa é (a), 1.

   Dessa vez, o valor de `i` será usado no `if` e só depois incrementado. Como o resto de *0* dividido por qualquer número também é *0*, o `for` só executa uma vez. Mas o valor de `i` ainda será incrementado, imprimindo o valor de *1*.

3. Vai imprimir 3.

   O `for` externo vai contar de 0 a 5, mas dentro do `for` tem um `if` que pré-incrementa o `i`. Esse `if` vai quebrar o *loop* no momento em que o valor retornado for divisível por 3, isto é, quando `i` valer 3 nesse caso.

5. A resposta certa é (f). A segunda linha do método `main` não compila pois estoura o limite de byte.

6. A resposta certa é (1). O código não compila pois não podemos declarar um `char` negativo.

7. A resposta certa é (f). O código compila e imprime um outro valor (65 + 3).

8. A resposta certa é (b). O código compila e joga uma exception por causa da divisão inteira (são `ints`) por zero.

9. A resposta certa é (c). O código compila e imprime positivo infinito. A precedência de operadores é primeiro a divisão, por isso compila.

10. A resposta certa é (a). Não compila, não há comparação entre `boolean` e números.

5.2 - Exercícios

1. A resposta certa é (a). Não compila pois o resultado do parenteses é uma `String` que não possui o operador de divisão.

2. A resposta certa é (b). `true==false` é `false`. O inverso disso é `true`. Concatenando com `""` o resultado é `true`, que é igual a `true`. Portanto, o operador ternário devolve `1` que é diferente de 0, imprimindo `false`.

5.3 - Exercícios

1. A resposta certa é (b). O código compila e imprime `true` e `false`.

2. A resposta certa é (d). Compila e imprime `false`, `false`, uma vez que a String 2 vale `s`.

3. A resposta certa é (c). Compila e imprime `true`, `true`. Por mais que `substring` devolva uma nova String, nesse caso ele devolveu a String inteira, a própria String.

4. A resposta certa é (c). O código imprime `true` e `true`. Note que o método `equals` não foi sobrescrito.

5. A resposta certa é (a). O código não compila pois `D` não é do tipo `C`.

5.4 - Exercícios

1. A resposta certa é (c). O código compila normalmente e imprime `0` caso não seja passado nenhum argumento.

2. A resposta certa é (a). O código não compila pois a variável `valor` é `final` e não pode ser alterada, mas tentamos efetuar uma atribuição dentro do `if`.

3. A resposta certa é (a). O código não compila pois tenta atribuir 15 a uma variável e conferir o valor 15 como se fosse um `boolean`.

4. A resposta certa é (a). O código não compila pois não existe palavra-chave `elseif`. Devemos fazer um `else if` para compilar.

5. A resposta certa é (b). O código não compila pois o `else` não está aplicado ao `if`: para ser aplicado ao `if`, ele deve vir imediatamente após seu bloco. Como o `if` não possui chaves, somente a primeira linha pertence a ele.

5.5 - Exercícios

1. A resposta certa é (d). Ao rodar com 5 argumentos, o código imprime `mais argumentos`.

2. A resposta certa é (a). O código não compila pois `tamanhoEsperado` não é uma constante. Somente podemos verificar `case` de `switch` em variáveis finais inicializadas diretamente.

3. A resposta certa é (b). A `String "42"` é uma `String` uma vez que ela está entre aspas. Portanto, o código imprime `Guilherme`.

4. A resposta certa é (a). A sintaxe do `case` é com `:` e não com `{`, o código não compila.

5. A resposta certa é (a). O código não compila pois o `case` não aceita expressões como `< x`, mas sim um valor definido em tempo de compilação.

6. A resposta certa é (a). O código não compila pois há código que não será executado após `break`.

6.1 - Exercícios

1. A resposta certa é (f). Não faz sentido ter colchetes antes da declaração do tipo, portanto `[]int x` não compila.

2. A resposta certa é (b). A segunda linha não compila pois, ou você passa o tamanho, ou passa os valores.

3. A resposta certa é (x). O programa não compila pois a segunda e a terceira linha tentam redefinir uma variável já definida. Caso o nome da variável seja corrigido, o código compila e imprime nada ao rodar (um array pode ter tamanho zero).

4. A resposta certa é (c). O programa inicializa `i` para o tamanho do array, acessando uma posição inexistente. Portanto dá erro em execução (exception).

5. A resposta certa é (b). Não tenha medo de simular o código na mão. Simule a memória e perceba que dá uma `Exception`.

6. A resposta certa é (f). Não tenha medo de simular o código na mão. Simule a memória e perceba que o resultado é `2, -5`. Para isso, desenhe os três espaços do array, aponte os valores iniciais o e continue atribuindo valores, executando o código.

   Durante a prova, simular os arrays e os ponteiros é ideal para não se perder em códigos complexos de referências e valores com arrays.

7. A resposta certa é (d). Compila e imprime `true`: note que não existe criação de um novo array, nós simplesmente temos duas referências (`valores` e `vals` para o mesmo array na memória).

8. As respostas corretas são (a), (b), (f) e (j).

6.2 - Exercícios

1. A resposta certa é (a). Não compila pois, ao inicializarmos o array `zyx`, utilizamos um array de uma única dimensão.

2. A resposta certa é (f). Na posição 2, temos o array `z`, que tem 30 casas, portanto temos o resultado `30`.

3. A resposta certa é (g). O código compila e imprime `30` normalmente. Não há problema algum em apontar para um novo array.

4. A resposta certa é (a). Nesse exemplo, é guardado um valor `double` em uma das posições do array de `int idades`. Isso está incorreto portanto não compila.

   Na declaração do array de duas dimensões `tabela`, são informados os tamanhos das dimensões. Errado pois os tamanhos devem ser definidos na inicialização e não na declaração.

   Na inicialização do array `cubo`, não foi colocado o tamanho de nenhuma das dimensões.

6.3 - Exercícios

1. A resposta certa é (a). O código não compila, pois a classe `ArrayList` não foi importada.

2. A resposta certa é (d). O código roda e imprime `true`, pois foi removido um elemento da lista.

3. A resposta certa é (f). O código roda e imprime 1 pois ele remove o primeiro elemento igual ao elemento passado.

4. A resposta certa é (h). O código compila e imprime 5.

5. A resposta certa é (a). O código não compila, pois o método `toArray` sem argumentos retorna um array de `Object`.

6. A resposta certa é (b). O código inclui os elementos sempre no final da `ArrayList`, portanto imprime `a` e depois `d`.

7. A resposta certa é (a). O código não compil,a pois a ordem dos parâmetros para o método `add` é `int, String`.

8. A resposta certa é (b). O código compila e imprime somente `a`. Isso porque ele executa um `next` durante o passo de iteração do laço `for`, o que acaba consumindo o segundo elemento sem imprimi-lo.

9. A resposta certa é (c). O código compila e imprime `a, b, c, d`, pois o laço está alterando o valor referenciado pela variável `s`, e não o valor contido dentro da nossa `ArrayList`.

7.1 - Exercícios

1. A resposta certa é (e). O código compila e ao rodar, `a` não é maior que `100`, portanto imprime `10`.

2. A resposta certa é (c). O código já compila pois a variável não é final, e entra em loop infinito.

7.2 - Exercícios

1. A resposta certa é (a). O código não compila, pois o laço nunca é quebrado e nunca chega a executar o código que imprime `b`.

2. A resposta certa é (a). O código não compila, pois o código dentro de `for` nunca será executado.

3. A resposta certa é (a). O código não compila, pois tentamos definir o tipo de duas variáveis no nosso `for`.

4. A resposta certa é (b). Compila e imprime os valores 0 até 9.

5. A resposta certa é (a), O código não compila pois a condição de um `for` deve ser única.

6. A resposta certa é (e). O código compila e imprime `0 1 1 2`

7.3 - Exercícios

1. A resposta certa é (b). O código compila e imprime `false`, pois ele sempre entra no laço pelo menos uma vez.

2. A resposta certa é (b). Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Não imprime nada caso 3 a 9 argumentos. Imprime 'Finalizou' caso 10 ou mais argumentos.

3. A resposta certa é (c). O código compila e imprime 'o', já que a condição é 'i' maior que 10.

4. A resposta certa é (a). O código não compila, pois faltou um ponto e vírgula.

5. A resposta certa é (c). Compila e sai.

7.4 - Exercícios

1. A resposta certa é (a). Quando iteramos por duas coleções ao mesmo tempo, podemos usar tanto o `for` quanto o `while`, mas o `for` é mais simples, pois passa por todos os elementos já com a inicialização e incremento bem definidos dentro do laço.

2. A resposta certa é (a). Usamos o `for` tradicional (com ou sem `Iterator`) para remover elementos. Poderíamos usar o `while`, mas ele não está na lista de respostas.

3. A resposta certa é (b). Devemos usar o `do...while`, que garante a execução pelo menos uma vez do código.

4. A resposta certa é (d). `for`, `while` ou `do...while` resolvem o problema, mas o mais comum é o `while`.

5. A resposta certa é (b). Não é possível inicializar os valores de um array com o `enhanced for`, portanto usamos o `for`.

7.5 - Exercícios

1. O código não compila pois o segundo `if` está fora do bloco do for e tenta acessar uma variável definida dentro dele. Lembre-se que o escopo de um bloco `for` sem chaves é uma única instrução, no caso o primeiro `if else if`.

2. A resposta certa é (d). Compila e imprime 0 até 19, 21 até 24, 26 até 29.

3. A resposta certa é (j). Compila e ao rodar com o argumentos imprime 1 até 15, fim.

8.1 - Exercícios

1. A resposta certa é (a). O código não compila pois existe um `return` sem valor.

2. A resposta certa é (a). Não compila, pois não basta a variável ser final; para funcionar no *switch*, ela tem que ser definida com valor constante (sempre 5, por exemplo).

3. A resposta certa é (b). O código não compila porque o método `c` retorna um `long`, e esse `long` é utilizado como retorno no método `a` e no método `b`. Ambos precisam de um retorno do tipo `int`, que não tem conversão automática.

4. A resposta certa é (a). O código não compila, pois não existe retorno de método com dois valores como `int, int`.

8.2 - Exercícios

1. A resposta certa é (a). O código não compila pois os métodos não possuem tipo de retorno definido. Típica pegadinha: parece focar em `static` mas está focado em outra coisa.

2. A resposta certa é (b). Imprime x, depois y.

3. A resposta certa é (d). Compila e imprime `z`.

4. A resposta certa é (a). O código não compila pois tenta acessar `this` dentro de um contexto estático.

5. A resposta certa é (b). O código compila e imprime `x` e `y`.

8.3 - Exercícios

1. A resposta certa é (a). O código não compila, pois não há sobrecarga de método ao alterar só o retorno.

2. A resposta certa é (c). Compila e imprime `15`, `15` e `15.0`.

3. A resposta certa é (a). O código não compila.

4. A resposta certa é (a). Não compila: os métodos não são estáticos.

5. A resposta certa é (d). Compila e imprime `3`.

6. A resposta certa é (b). Compila e imprime 1.

7. A resposta certa é (a). Não compila, pois tem várias variáveis locais (parâmetros) com o mesmo nome.

8. A resposta certa é (b). Compila e imprime 1.

8.4 - Exercícios

1. A resposta certa é (d). O código compila e joga um `NullPointerException`.

8.5 - Exercícios

1. A resposta certa é (a). O código não compila por causa do loop, quando um construtor de um tipo chama outro construtor do mesmo tipo em loop direto.

2. A resposta certa é (d). O código não compila pois as classes definem parênteses a mais. Cuidado.

3. A resposta certa é (f). O código compila e não imprime nada.

4. A resposta certa é (f). O código compila e não imprime nada.

5. A resposta certa é (e). O código compila e joga exception ao entrar em loop infinito.

8.6 - Exercícios

1. A resposta certa é (a). Ocorre um erro de compilação na classe `Teste` ao tentar chamar o construtor com acesso `default` de outro pacote.

2. A resposta certa é (c). Compila e imprime `3`.

3. A resposta certa é (b). Não compila na declaração do método `private public`.

4. A resposta certa é (a). O código não compila, pois a classe `A` é a própria classe do método `main`, e ela não tem método `a`.

5. A resposta certa é (b). Imprime `1`, pois o método que recebe `String` não está visível no pacote principal.

6. A resposta certa é (a). Não compila, pois a palavra `default` não pode ser usada como modificadora de visibilidade de método (ela é usada a partir do Java 8 de outra maneira, que não é cobrada nesta prova).

7. A resposta certa é (b). O código compila e imprime 1

8.7 - Exercícios

1. A resposta certa é (b). Compila e imprime 0, são duas instâncias de `B`!.

2. A resposta certa é (c). Compila e imprime 5.

3. A resposta certa é (b). Compila e imprime 0, existe *shadowing* aqui no `setter`.

4. A resposta certa é (c). Compila e imprime 5.

5. A resposta certa é (d). Compila e imprime 10, existe *shadowing* aqui no `setter`, então não há problema de a variável ser `final`.

8.8 - Exercícios

1. A resposta certa é (a). Não compila pois somente variáveis podem ter aplicadas *auto increment* e decremento.

2. A resposta certa é (c). Compila e imprime 151.

9.1 - Exercícios

1. Aqui não ocorre sobrescrita. Como os parâmetros são diferentes, ocorre uma sobrecarga (não confundir na prova *overload* com *overwrite*). Ou seja, é um *overload* com herança.

2. A reescrita é válida, pois `FileNotFoundException` é subclasse de `IOException`.

3. A resposta certa é (a). O código não compila pois há um ciclo na herança.

4. A resposta certa é (a). O código não compila pois usa herança múltipla de classes, que não existe em Java.

5. A resposta certa é (c). O código compila e não imprime nada.

6. A resposta certa é (a). O código não compila pois não existe construtor de `B` ao qual `A` tenha acesso para herdar do mesmo.

7. A resposta certa é (a). O código não compila pois `A` não tem acesso a variável de `B`.

8. A resposta certa é (b). Compila e imprime 't'

9.2 - Exercícios

1. A resposta certa é (a). O código não compila pois faltou o `import` de `java.io.*`.

2. A resposta certa é (b). O código compila e imprime 'b'.

3. A resposta certa é (c). O código compila e imprime 'c'.

4. A resposta certa é (a). O código não compila: `C` não possui método que receba `double`.

5. A resposta certa é (a). Não compila, pois interface não pode ter método com corpo da maneira como foi definido aqui.

6. A resposta certa é (d). Compila e imprime 'd'

7. A resposta certa é (g). Compila e entra em loop.

8. A resposta certa é (b). Compila e imprime 'b'.

9. A resposta certa é (a). Não compila, não existe `super.x` na classe `B`.

9.3 - Exercícios

1. A resposta certa é (a). Não compila pois há um erro de *copy* e *paste* nos nomes das variáveis.

2. A resposta certa é (c). Compila e imprime 2.

3. A resposta certa é (c). Compila e imprime 2.

4. A resposta certa é (c). Compila e imprime 2.

5. A resposta certa é (b). Compila e imprime 1.

6. A resposta certa é (a). O código não compila pois o método `fecha` não é público.

7. A resposta certa é (d). O código imprime 'fechando conta normal'.

8. A resposta certa é (a). O código não compila pois, ao sobrescrevê-lo, tentamos definir um escopo menor. Não compila também porque o método `fecha` é *package protected* dentro de `Conta`.

9. A resposta certa é (c). O código compila e imprime 'fechando financeiro'.

10. O código compila normalmente. Apesar de o método `ligar` não estar declarado na classe `CarroConcreto`, a classe herda este método de `Carro`, logo, não é necessário reescrevê-lo (poderia reescrever se achasse necessário).

    A declaração de que `CarroConcreto` implementa `Veiculo` também não era necessária, pois `Carro` já implementa `Veiculo` e `CarroConcreto` é um `Carro`.

9.4 - Exercícios

1. A resposta certa é (c). O código compila e roda, ao rodar não dá exception.

2. A resposta certa é (d). O código compila e roda, ao rodar dá exception.

3. A resposta certa é (a). O código não compila: `D` até implementa `Z` e `W` mas não implementa `Y`.

4. A resposta certa é (d). O código compila: algum subtipo de `D` pode implementar `Y`. Ao rodar, ele dá exception.

5. A resposta certa é (c). Compila, pois apesar de `B` não implementar `Z`, um subtipo dele pode (e na prática já o faz) implementá-lo. Ao rodar não dá exception nenhuma.

6. A resposta certa é (d). O código compila e roda dando exception.

7. A resposta certa é (a). `D` não implementa `Y`, não compila.

8. A resposta certa é (b). O código não compila: `instanceof` é minúsculo.

9. A resposta certa é (c). O código compila e imprime `true`.

9.5 - Exercícios

1. A resposta certa é (c). O código compila e imprime 2.

2. A resposta certa é (a). O código não compila.

3. A resposta certa é (a). O código não compila.

4. A resposta certa é (a). O código não compila: não faz sentido acessar o } `super` de outro objeto que não eu mesmo.

5. A resposta certa é (c). O código compila e não imprime nada: o método definido não é um construtor!

6. A resposta certa é (a). Não compila pois tentamos invocar dois `this`.

7. A resposta certa é (c). Compila e não imprime nada.

8. A resposta certa é (c). Compila e não imprime nada.

9. A resposta certa é (a). Não compila: não podemos referenciar um método de instância ao invocar um construtor `this`.

10. A resposta certa é (b). Compila e imprime ‘2‘.

9.6 - Exercícios

1. A resposta certa é (c). Compila e imprime ‘2‘.

2. A resposta certa é (a). A classe `B` não compila.

3. A resposta certa é (b). Compila e imprime ‘1‘.

4. A resposta certa é (b). Compila e imprime ‘1‘.

5. A resposta certa é (a). Não compila, pois o método é final.

6. A resposta certa é (c). Compila e imprime ‘b‘.

10.1 - Exercícios

1. A resposta certa é (d). A única exception da lista que não é checked é a `IndexOutOfBoundsException`

10.2 - Exercícios

1. (b) e (e) são corretas. (a) está errada pois podemos usar exceptions mesmo sem entradas do usuário. (c) está errada pois podemos manter o programa rodando mesmo que uma exception ocorra. (d) está errada pois devemos usar outras estruturas para controlar o fluxo de nosso programa, como `if` por exemplo.

2. (a) e (c) estão corretas. (b) e (d) estão incorretas por serem os opostos das certas, e (e) está incorreta pois exceptions não são maneiras de aumentar a segurança.

10.3 - Exercícios

1. A resposta certa é (a). O código não compila pois a variável local nunca foi inicializada.

2. A resposta certa é (c). Quando ocorre a exception, o fluxo desvia para imprimir "b" e depois continua normal com o "c".

3. A resposta certa é (c). Quando ocorre a exception, o fluxo desvia para imprimir "b", passa pelo `finally` imprimindo "c", e depois continua normal com o "d".

10.4 - Exercícios

1. A resposta certa é (a). Devemos colocar uma `java.io.IOException`.

2. A resposta certa é (b). O código compila pois ele cria um array de dimensão 2. Ele imprime `acefdb`.

3. A resposta certa é (c). O código compila e imprime `ace`, jogando uma `Exception`.

4. A resposta certa é (a). O código não compila pois o método `m2` deve tratar ou jogar `java.io.FileNotFoundException`.

5. A resposta certa é (a). O código não compila pois o método `main` deve tratar ou jogar `java.io.IOException`.

6. A resposta certa é (c). Compila, imprime `ace` e joga a `Excepion`.

7. A resposta certa é (e). Compila, e imprime `acedb`.

8. A resposta certa é (b). Compila, e imprime `acefdb`, note que não jogamos a exception, somente a instanciamos.

9. A resposta certa é (a). Não compila: o `System.out` do `f` é `unreachable`.

10. A resposta certa é (a). Não compila: a palavra `throw` deveria ter sido usada para jogar a Exception.

11. A resposta certa é (c). Compila, e imprime `ace` e estoura uma `Exception`.

10.5 - Exercícios

1. A resposta certa é (e), `StackOverflowError`.

2. A resposta certa é (d). `OutOfMemoryError`, pois tem um loop infinito.

Índice Remissivo